ERC+: an object+relationship paradigm for database applications

Stefano SPACCAPIETRA

Ecole Polytechnique Fédérale, DI - Laboratoire Bases de Données,
IN - Ecublens, 1015 Lausanne, Switzerland
Telephone: +41 21 6935210 Fax: +41 21 6935195
email: spaccapietra@di.epfl.ch

Christine PARENT, Marcos SUNYE, Kokou YETONGNON

Université de Bourgogne, Département Informatique, B.P. 138,
21004 Dijon Cedex, France
Telephone: +33 80395891 Fax: +33 80395815
email: badine@satie.u-bourgogne.fr

Antonio Di LEVA

Dipartimento di Informatica, Universita degli Studi di Torino,
Corso Svizzera 185 - 10149 Torino, Italy
Telephone: +39 11 7712002 Fax: +39 11 751603
email: dileva@di.unito.it

Abstract

The concepts of object oriented data models aim towards modeling of application objects close to the user's view. Yet developers of applications relying on object oriented database management systems are facing problems resulting from the limitations of object oriented data models to describe adequately the full range of possible associations between objects and between processes.

This paper focuses on the conceptual data modeling needs of object oriented database applications, and proposes an object+relationship model, ERC+, which meets database application requirements by merging traditional semantic data models features with object oriented capabilities such as structural object orientation, inheritance, and object identity. It is an extended entity-relationship model specifically designed to support complex object description and to allow multiple perceptions of objects. The ERC+ model provides the foundations for an integrated environment of tools called SUPER that has been designed to aid the development of database applications. The key features of the SUPER environment include: a graphical schema editor for describing complex objects, a data browser and a graphical editor for formulating ERC+ queries.

Keywords

Object-oriented modeling, object-oriented databases, application requirements, visual programming, user interfaces, entity-relationship model, information systems

1. Introduction

Object orientation has rapidly become very popular. It has, indeed, proved to be a powerful and practical programming paradigm by bringing many benefits such as modularity, reusability, flexibility, and extendibility to designers and developers of software systems. Implementors of large and complex systems have found the object-oriented (OO) modeling paradigm particularly appealing for the following reasons:
- it uses a few but powerful concepts, including: abstract data types, inheritance mechanism, and object identity; the result is a clean abstract paradigm and methodology based on sound foundations of programming languages;
- it stresses modularity in the design of complex systems by transforming them into a collection of interoperable objects, and thus allows developers to react quickly and easily to evolution of specifications;
- it provides the inheritance mechanism to enforce reusability of both data structure descriptions and operation definitions; this ultimately decreases software development time and cost.

Taking advantage of the above features, software producers have built complex systems such as: software engineering environments, CASE tools, OO languages, OO interface systems, OO database management systems (DBMS), etc. The benefits of the OO approach have found good acceptance in the database markets.

This success story raises the following question: can domains other than large software system design get the same benefits by adopting the OO modeling approach? Can database applications benefit from it? Modern applications in data intensive fields such as computer aided design/computer aided manufacturing (CAD/CAM), office automation, computer integrated manufacturing (CIM), robotics, geographic systems, must deal with highly structured and interrelated information objects. These applications have the same complexity as large software systems, and therefore, OO technology has naturally been advocated as a key to their efficient design and implementation. Hopes are high that OO DBMS will significantly improve over traditional database technologies in meeting the requirements of these applications. In particular, systems supporting data structures which allow database objects to look like those users deal with in the real world will make user-DBMS interaction much easier. It will avoid what is presently experienced as the impedance mismatch problem (i.e. the problem of having two interacting agents reasoning with different paradigms).

Unfortunately, OO technology applied to information systems has not, so far, completely fulfilled these expectations. Indeed, when moving a technology from one domain to another one, existing concepts have most often to be reshaped to cope with the new environment, and new concepts have to be added to achieve full power with respect to the new goals. As an example of the former, as user's profile changes from system programmers to application designers and end users, some OO features may place an additional burden on end users. Data encapsulation of objects, for instance, is a highly desirable feature for efficient design management at the implementation level. It allows to decompose the design process and let each designer independently develop his/her part. Database practice, on the contrary, is based on the principle of open access to information to everybody. Therefore, encapsulation needs to be partly traded off for easy access to attributes, so that general purpose query languages (SQL-like or others) may be implemented.
On the other hand, the OO paradigm needs additional concepts to achieve full representational power (namely, to acquire adequate concepts for conceptual representation of associations among objects, as discussed in detail in section 2). Indeed, the descriptive power of the data model is of major importance in the database field. While system programming primarily aims at efficiency, database designer's primary concern is on accurate representation: elaborating a description of the reality of interest which captures as much as possible of the semantics of data, independently of any implementation consideration, and which is as close as possible to the user's perception. This activity is called conceptual design. The elaboration of an implementation oriented representation of data and of performance related specifications is called physical design, and is dealt with in a separate design step which follows the conceptual design.

This paper is precisely devoted to bridging the gap between the OO paradigm and its successful usage for the development of database applications. Our starting point is to contrast the features of object orientation with the requirements of developers of database applications. Based on this analysis, we forward a proposal on those issues where a deficiency has been identified. Basically,

these deficiencies are: inadequate representation of associations, lack of proper support for generic data manipulations, no concept for the description of the global application behavior. Extending the OO data model results in what is nowadays known as an object+relationship (OR) modeling approach. OR models intend to combine the advantages of OO models with those provided by semantic data models. For the description of the behavior of an information system, traditional research in database modeling offers Petri net-like approaches which can be used in conjunction with an OO or OR data model.

We first deal with the issue of conceptual modeling of data structures (objects and associations). A structural data model, called ERC+, is proposed. ERC+ is an extended Entity-Relationship (ER) model [Chen 76], specifically designed to support objects with a complex structure. To some extent it is similar to recently proposed OR models [Rumbaugh 87, Albano 91]. It provides full support for explicit conceptual description of generic relationships among objects. Second, we discuss data manipulation languages (DML), which complement the data model to provide for full and consistent capabilities for user-DBMS interaction. The ERC+ algebraic query language is presented here. Together with an equivalent calculus [Parent 90] they form the theoretical background for the development of user oriented languages. For instance, an ERC+ SQL-like DML has been developed [Sunye 92]. Third, we present the graphical query language which has been implemented as part of a graphical environment, called SUPER, to aid the design and development of database applications. SUPER is based on the ERC+ approach. It presently includes tools for graphical description of database schemas and for graphical data manipulation, either through schema or data browsing or through query formulation. SUPER ensures user interaction with both OO DBMS and traditional relational DBMS, thus providing for DBMS independence. Last, the paper shows how ERC+ may be coupled with a Petri net based process model to express the global dynamic behavior of the information system. While OO methods allow the description of local behavior of objects, the process model takes into account concurrency and synchronization problems of object messages to represent the sequence in which operations are performed .

The remainder of the paper is organized as follows. In the next section we briefly examine the requirements of database applications and analyze the OO approach from the data modeling perspective. The characteristics of an enhanced model, to meet application requirements, are introduced. In section 3 a detailed and formal description of the proposed ERC+ model is given. Sections 4 and 5 discuss the issues involved in using algebra-based data manipulation languages for defining, accessing, and managing complex objects databases according to the ERC+ paradigm. The description of the SUPER environment is given in section 6, which presents its graphical schema editor, and section 7 which discusses its graphical query editor. Section 8 considers the requirements for describing the dynamics of database applications and shows how a process model of dynamics can be integrated with the structural description capability of ERC+. Finally, section 9 concludes the paper.

2. Modeling database applications

In the database design area, the major concern in conceptual modeling of application objects is to create representations which are close to reality, and independent from implementation issues. Conceptual modeling is understandable by users (they need not to know about peculiarities of DBMS systems). It therefore makes easier to check that the resulting database schema meets users requirements. It improves the chances of a correct design, by separating representational issues from implementation issues. It allows to change the implementation without having to modify the conceptual schema.

Section 2.1 discusses requirements of database applications for conceptual modeling of both static (data structures) and dynamic (behavioral) aspects. The extent to which traditional data models fulfill these requirements is briefly surveyed in section 2.2. A similar evaluation for OO models is the subject of section 2.3. The last section, 2.4, introduces the characteristics of an OR model which is intended to improve over OO models towards fully supporting application requirements.

2.1. Applications modeling requirements

Database applications are characterized by their need to represent, relate and manipulate objects with complex information structures. Typically, objects having the same properties are collected into

classes which are described by object types. Attributes are used to represent the properties of the objects in a class. The attribute structure of an object can be described by a tree whose root is the object, whose branches are of variable and unrestricted length, and whose nodes represent the components of the object. Such objects are usually called **complex objects**, to contrast them with the simple flat objects supported by the relational approach (the tuples). Each object has a unique object identity (oid), and a composite value. The object value is composed from the atomic values attached to the leaves of the tree, according to the structure of the set of attributes attached to its object type. The object identity is system defined (not visible to users) and allows to denote objects independently from their value. From the manipulation point of view, it is essential that objects may be manipulated as a whole logical unit, irrespectively of their complexity, and accessed through any of their components. Examples of complex object types are given in figure 2.1 and 3.1.a.

As the same objects are shared by different applications, they may be perceived by these applications in different ways. Facilities for allowing multiple perceptions of objects are therefore essential. This includes the possibility for each application to attach its specific set of properties to an object, while the database keeps the knowledge that the object is unique despite its different representations. Most important is the ability to define different classification schemes, i.e. different object types and associations in between. For example, let us consider a database for the management of a garage. From the receptionist's point of view, cars may be perceived as single objects, grouped in a class described by the Car object type, with properties including the description of the engine and body of the car (see figure 2.1). From the mechanic's point of view, engines and bodies may be perceived as independent objects and described as such in Engine and Body object types. Despite their difference, both points of view are modeling the same reality. The extent to which a modeling approach supports multiple points of view defines the degree of freedom, for an application, to define its own perception of the database without having to comply with the perceptions of other applications sharing the same database.

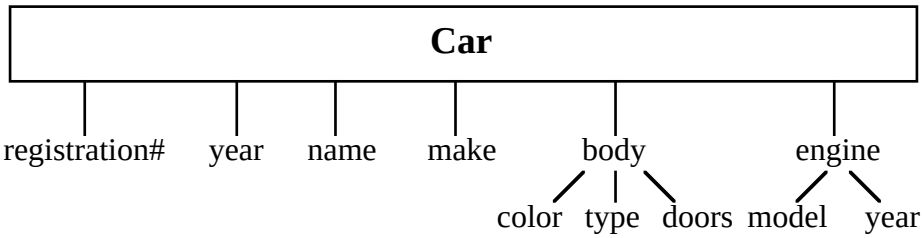


Figure 2.1: The receptionist's view: one object type, Car
(boxes represent object types)

Supporting multiple perceptions on complex objects is but the first step in representational power. Objects do not live in isolation; they are related to, and interact with, other objects. For instance, if the above garage management application is to deal also with car owners as customers, the database will include a Customer object type, whose objects will be associated to Car objects to express the ownership association. Research on semantic data models has shown that there are several kinds of associations which are candidate for inclusion in a data model. They range from generic associations, whose semantics is known only to the application, not to the DBMS, to associations having a specific predefined semantics: the component (part-of) association, the generalization (is-a) association, etc. [Brodie 84]. The above car ownership association is an example of the generic type. Existing data models differ in the support of association types, as they differ in the support of complex objects.

Objects, properties and associations characterize the capability of a data model to describe data structures. With respect to dynamic modeling, application requirements command capabilities to describe rules governing the behavior of objects as well as rules governing the behavior of applications. The former express how an object may evolve with time and what operations may be performed on it. Each object type bears such a description. The latter allows the description of interactions among objects and is typically based on the concepts of event, condition and action: for each event to which the database should react, a rule describes the pre- and post-conditions attached to the event and the corresponding actions to be performed. Object-specific operations also play an important role in supporting data manipulation requirements. However, they need to be complemented with generic query and update languages to support generic and ad-hoc manipulations.

2.2. Traditional data models

Current relational DBMS poorly respond to the above requirements. They only support flat data structures: relations, whose components are restricted to atomic value attributes (this restriction is known as the first normal form rule). An object in a relation (a tuple) is a list of atomic values. Composite attributes are not allowed: for instance, it is not possible to specify a date attribute as composed of day, month and year attributes. Multivalued attributes are not allowed: for instance, it is not possible to associate several values to a phone# attribute. The description of an object cannot be expressed as a self contained block of information, it is instead spread over several relations. A major hindrance of such a solution is that the resulting representations do not parallel the structure of the objects being modeled as perceived by users. Hence the problems users have in understanding a relational representation and in manipulating a relational database.
Moreover, relational DBMS ignore associations. These have to be represented and managed by applications. Only the most recent versions of some major relational DBMS include a limited support for associations in the form of a capability to specify referential integrity constraints and have them checked by the system. As far as dynamic aspects are concerned, no specification mechanism is provided by relational DBMS. Dynamics is entirely within application programs. Inversely, relational DBMS support a variety of data manipulation languages: the relational algebra and calculus, as theoretical basis, SQL and QUEL as textual user oriented DML, and QBE as graphical language. Using DML functionality's, users may define their own view over the relations in the database schema. Multiple perceptions of the same objects are thus easily defined.

Many extensions or alternatives to the traditional relational data model have been proposed in order to: better capture the semantics of the real world and allow the representation of complex and/or composite objects. The most notable of these models are nested relational models (also known as NF2: non first normal form) and semantic data models. Nested relational models [Abiteboul 89] partially meet applications requirements by admitting relation-valued attributes: the value of an attribute in a tuple may be a set of tuples. This in essence relaxes the relational first normal form restriction. Relation-valued attributes can be accessed and/or retrieved in the same manner as relations. However, with respect to complex object management, NF2 models still have a major restriction in the fact that the structure of an object is purely hierarchical: sharing of components is not supported and no cycles are allowed in the structure of an object (a component cannot be of the same type as the composed object). Furthermore, facilities for restructuring a nested relation to define a different point of view are limited. Thus, NF2 models are not fully appropriate for representing complex objects and multiple perceptions. They also ignore dynamic aspects.

The goal of semantic data models has been to represent the semantics of the real world as closely as possible. However, the emphasis put on data description has not found an equivalent counterpart in the data manipulation area. Semantic models are therefore mainly used in the initial phase of database design to produce a conceptual representation of the future database. This initial representation is then translated into a lower level target model (e.g. relational, OO, etc.) to be implemented onto a DBMS. An example of semantic data model is provided by extended ER models, which make use of the concepts of entity, relationship, attribute, generalization and constraint to closely represent the properties of real world objects and associations in between. An entity is the database representation of an object, with all its information attached: a complex object can be represented as a single entity. Relationships represent generic associations among two or more objects. Generalization is a specific association type used to specify that entities in one class (the sub-class) are also represented in another class (the super-class). For instance, a generalization link (also called an is-a link) associating a Sport-car entity type to a Car entity type specifies that each object in the former class also belongs to the latter class (i.e., each sport-car is a car).
The ER approach has not developed concepts for the description of behavioral aspects. Dynamic rules may be expressed using associated constraint specification languages (usually based on first order logic), but no provision is made for specification of object specific operations. From the data manipulation perspective, a few languages have been proposed but none of them has been implemented in a commercial DBMS. As for support of multiple perceptions, this is achieved to a great extent [Spaccapietra 92], but work remains to be done, as in the NF2 case, on restructuring operations.

2.3. OO modeling

In the database area, the OO approach is the first significant attempt to define a data model which simultaneously takes into account both structural and behavioral specifications. As such, it definitely improves the state of the art in database modeling. We examine here how it matches application requirements.

From the data modeling perspective, the key features of the OO approach are: object identity, complex data structures, abstract data types and inheritance. The concept of object identity (oid) is used to associate each object with a unique identity. This oid distinguishes the object from other objects without having to rely on user defined values. This definitely corresponds to what applications need as denotational mechanism. User defined identifiers (the "unique key" attributes) cannot safely be used as oids because they confuse identity and data values [Khoshafian 90]. For example updating an attribute which is (part of) a key is not allowed even though it is a value attribute with a specific meaning as other value attributes.

Objects may be as complex as needed in an OO schema. OO models allow unrestricted iterative decomposition of an object into components. Moreover, they allow a component of an object to be an object itself (i.e., with its own identity) rather than a value attribute. Component objects may be of the same type as the object they are part of. Objects whose composition includes other objects are hereinafter termed **composite objects**. Conversely, **atomic object** will denote an object whose components are all value attributes. This introduces a clear difference between OO models, supporting complex and composite objects, and extended ER models, which support complex objects but do not support composite objects. Allowing composite objects is the OO mechanism to support multiple perceptions of the same object. Every object can at the same time be an object on its own and serve as a component within other objects. This composition association is materialized by a reference attribute which points from the composed object to the component object. The domain of a reference attribute is the set of oids representing the objects referred to by the attribute. References are directed one-way links from a composite object to its components. For example, if the composition of a Car includes a Body and an Engine, where Car, Body and Engine are three object types, then this fact can be expressed through two reference attributes body and engine in the Car object type (see figure 2.2).
It should be noted that composite objects offer a limited flexibility in supporting multiple perceptions. For instance, it would not always be straightforward to add to the Engine object type a reference to Car to describe the cars which possess this engine, while ensuring that this reference holds a value consistent with the values in the (inverse) references from Car to Engine.

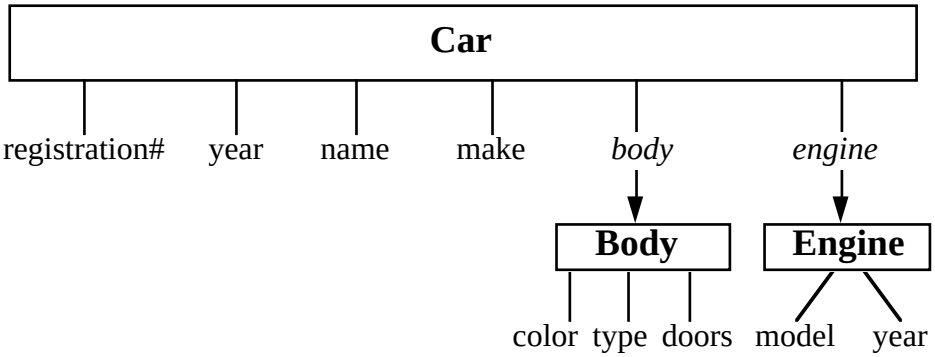


Figure 2.2: The global OO conceptual schema of the example garage database
(arrows with italic labels represent reference attributes)

Besides composition, OO models capabilities to express associations only include the inheritance link. An inheritance link between two object classes describes a sub-class / super-class relationship, which means that all objects of the sub-class are also objects of the super-class. It corresponds to the generalization association. Most often, the link is intended for reuse of structural (attributes) as well as behavioral (methods) definitions. The sub-type inherits these definitions from the super-type (downward inheritance). Thus, inheritance may be used as a mechanism for top-down design of a hierarchy of object types, where each design level refines the definitions (super-class) of the upper

level into a set of sub-class definitions. For instance, the description of a car can easily become established by refining the more general definition of a vehicle class. Refinement can be either in terms of adding new attributes or methods which are specific to cars and not valid for vehicles in general, or in terms of redefining existing attributes and/or methods attached to vehicle to make them more specific to cars.

Abstract data types convey the idea that the structural description of a set of objects (showing the properties they share) has to be complemented with the definition of the operations which can manipulate the objects. This allows to capture both static and dynamic aspects of objects. The definitions of the allowed operations (the methods, in OO terminology) represent the interface of the abstract data type. Essentially, this interface provides users with information on how to use the objects but not on how the objects are implemented. Furthermore, abstract data types use the concept of data encapsulation to separate implementation details from the usage of an object and to ensure that the only operations that can be performed on the objects are those specified in the interface.

While the above capabilities meet application requirements, they do not meet all of them. As the name says, OO main concern is on object description. From this point of view OO models do provide what is needed. On the contrary, their description of associations is far too restricted. There exist only two kinds of specific associations: composition and generalization. The main concern of OO programming systems is more on reusing the code segments that are used to implement properties and methods than on achieving an accurate description of real world objects. Composition and inheritance links are essentially used to avoid redundant specifications of data structures and operations. The concept of attribute is used as a unifying abstraction both for describing object properties (value attributes) and for embodying associations between objects (reference attributes). As Albano et al. pointed out [Albano 91], the main advantage of this unification is to simplify system implementation by using the same access mechanism to manipulate both attributes and associations. This trade off between implementation efficiency and conceptual capability reduces the expressive power of the model and penalizes designers of database systems.

In the database design area, a major concern is to create representations which are close to reality, and independent from implementation issues. This can not be achieved using only the composition and inheritance associations. To illustrate this, consider the already mentioned ownership association between the object types Car and Person. Using reference attributes (composition) to express this generic association may yield many possible representations, including the following ones:

1/ a reference from Person to Car,
2/ a reference from Car to Person,
3/ cross references between Person and Car,
4/ a new Ownership object is created to include a reference to Person and another one to Car,
5/ a new Ownership object is created which is linked by cross references to both Person and Car.

Composition is a directed association. It can not properly describe, at the conceptual level, a generic association which is non directed by definition. A car is not a component of a person, and vice versa. On the other hand, composition may be used to implement the generic association. It is, indeed, when implementation aspects are considered that a choice among the above solutions can be made on the basis of performance criteria and the available information on the most frequently used access paths.

Inadequate representation is not the only drawback of using reference attributes to express generic associations. It also does not allow to express declaratively semantic information attached to the different roles in an association. Cardinality constraints (how many cars a person may own? how many owners a car may have?) are an example of information which, in the OO approach, has to be coded in the implementation of the methods which are used to manipulate the objects, and is therefore hidden from the data description point of view. Second, the representation of the association is spread over the definitions of the objects participating in it. In some of the above solutions, the definition of the Ownership association is partially specified in each of the objects Person and Car. This does not promote the definition of application objects in an independent and incremental fashion: to establish a new association among existing objects may require the modification of the definition of these objects. Finally, reference attributes are not appropriate for expressing n-ary associations or associations which have attributes.

It has already been pointed out that, besides the major hindrances of using reference attributes to express associations, the OO approach fails in meeting applications requirements on two more issues. The first one is providing for generic data manipulation languages. To this extent, the encapsulation principle has been softened to allow the development of OO SQL-like languages. The second weakness is on description of global application dynamics: how application events are monitored by the system and how they have to be taken care of with respect to the evolution of the database. A new stream of research on so-called active databases is developing OO solutions to this problem. A more traditional approach is to combine the OO paradigm with some well-known mechanism (Petri nets, for instance) which has been proved to satisfy these requirements.

Enhancements to overcome the above limitations are examined in the next section. Corresponding proposals are forwarded in later sections.

2.4. Object+relationship models

Recently, researchers have attempted to make up for the above mentioned structural limitations of object orientation in database application by including constructs for representing associations and integrity constraints in object oriented models [Rumbaugh 87, Albano 91]. Rumbaugh et al have presented an extended object oriented modeling technique which they have used to support the conceptual design and implementation of software systems. In addition to the usual features of OO models, their model supports three types of relationships which have the same meaning as their semantic data model counterparts: generalization, aggregation, and generic association. However, the model limits all relationships to binary links between object classes. Another notion that is included in their model is the cardinality concept which is used to indicate the number of objects of one class that can be related to an object of the other class. Albano et al. have also proposed mechanisms for the inclusion of relationships and integrity constraints in a strongly typed object oriented database programming language. Their extensions to the OO model are more comprehensive than the proposal in [Rumbaugh 87]. They offer capabilities for expressing n-ary relationships and relationships which have their own attributes. Their model also incorporates mechanisms for expressing a large variety of integrity constraints. For example, inclusion and disjointness constraints can be defined between object classes. Referential constraints can be specified on a class to ensure that whenever an object is inserted into a class, all the objects used as components are elements of the corresponding classes.

Those extended OO models which include the concept of generic relationships are called object relationship (OR) models. In this paper we propose another approach for defining an OR model. By constrast to the proposal of Albano et al., which aims to embed relationship capabilities in a strongly typed OO programming language, we start with the well-known and largely favored by database designers, ER approach, which has specific capabilities for representing the semantics of relationships, and extend it to enable description of complex and composite objects. The success of the ER paradigm has many reasons: it is powerful but still simple (few concepts), its basic concepts are easily understood by users, database entities correspond to real world objects, database schemas may be illustrated with easy to read diagrams. By combining the ER paradigm with OO behavioral modeling concept (i.e. the attachment of methods to entity and relationship types), we complement a tool which has been designed for conceptual modeling with features it still lacks. This might be the most profitable way to achieve the best conceptual model database designers are looking for.

Extended ER approaches [Elmasri 85a, Parent 92] model real world objects with entity types. They use relationship types to model generic associations between two or more entities (objects). Like OO models, they provide an object identity for each entity and structural capabilities for the description of complex objects. In fact, regarding object modeling, the fundamental difference between existing extended ER approaches and the OO approach is that the ER paradigm restricts the components of an object to be attributes while the OO paradigm allows components to be objects. If objects of type B are part of objects of type A, then the ER representation will show two entity types, A and B, linked by a relationship R whose semantics will be known to the application as "B is component of A". This also holds for recursively composite objects, where a component of an object of type A is an object of the same type A. In other words, the composition association is not supported in ER and is replaced by a generic association (while OO, as stated above, does exactly the inverse).
The ER separation between objects and attributes relies on the assumption that, when a designer designates something in the real world as an object, whatever information (s)he wants to keep on this object, it has to be considered as a property of the object, hence represented as an attribute. The fact

that another designer classifies the same reality in some other way should not inhibit the first perception. Instead of using composite objects, the OR model we propose uses a more flexible mechanism to support multiple perceptions: derived entity types (also called virtual classes, or views). Similar to views in relational systems, a derived entity type represents objects which, through some derivation rule expressed in the OR data manipulation language, are built from other objects and relationships of the data base. Derived entity types model virtual objects in the sense that they do not add new objects to the database, but define a new point of view over existing information.

To illustrate how this mechanism supports multiple perceptions, let us consider again the garage example. The goal is to support both the receptionist's point of view (one object class, Car, with attributes including body and engine) and the mechanic's one (where Car, Body and Engine are object classes). The modeling process first represents by an entity type whatever is seen as an object class at least by one user. Next, relationships are defined among these entity types to represent generic and composition associations. The OR schema for the garage database will show three entity types, Car, Body, Engine, related by two relationships, Car-Body and Car-Engine. The attributes attached to each entity type are those defined by designers, except for those attributes which have been attached to a related "component" entity type. The Car entity type does not include information about the engine and body of the car (see figure 2.3). This information is available in the Engine and Body entity types, and may be accessed from the car entities via the relationships. The resulting structure forms the basic schema, the conceptual description of database objects. Derived entity types are added to the basic schema to represent objects which are perceived by designers differently from the structure in the basic schema. This is the case for cars as seen by the receptionist. A derived entity type, say RecepCar, is defined to merge in a single virtual entity type the information from the three basic entity types. Its schema is the one represented in figure 2.1. The derivation rule in this case specifies a relationship-join operation (see section 5). The receptionist will use the derived entity type for his/her interactions with the database.
Relationships and the derivation mechanism offer more flexibility than OO composite objects. If A and B are two related object types, it is easy to define a view where A has B as component attribute and a view where B has A as component attribute. For instance, if the ownership relationship links Person and Car, relationship-join operations may build both Person-with-cars (each person with the cars (s)he owns) or Car-with-persons (each car with its owners), depending on the order of the operands. This is harder to achieve relying on composite objects, because of the inherent direction of the composition link.

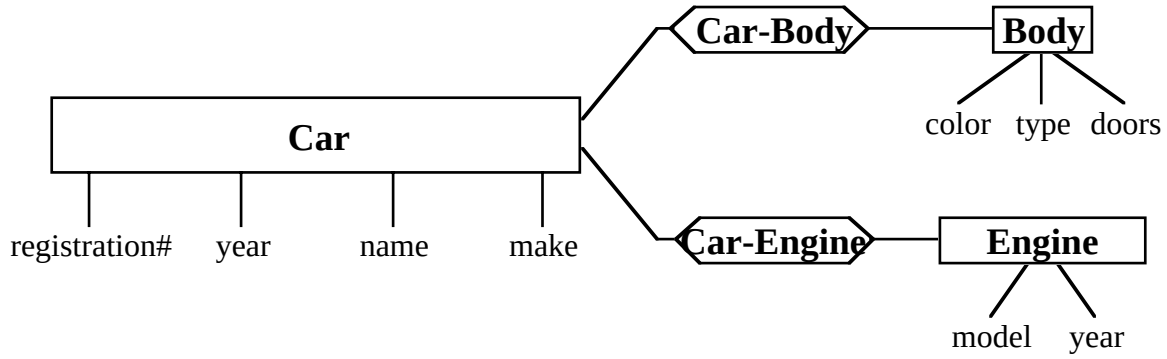


Figure 2.3: The garage conceptual schema described with objects and relationships
(diamond-shaped boxes represent relationship types)

The generalization association is also supported by OR models. The model we propose, ERC+, uses it to express sub-class / super-class relationships. However, it does not associate it with an implicit inheritance mechanism. Inheritance has to be asked for explicitly as part of the database manipulation. Again, this allows for more flexibility, by letting the user specify which inheritance (downward, upward), if any, (s)he wants. Moreover, ERC+ support another association type, called the may-be-a link, to express that objects may belong to two classes (which is called conjunction), without one class being a sub-class of the other. These enhancements to the OO inheritance link are discussed in detail in the next section.

3. ERC+: an object+relationship data model

ERC+ is an extended entity-relationship model, specifically designed to support complex objects. Its goal is to closely represent real world objects. It uses the basic concept of entity type to describe objects and the concept of relationship type to represent relationships among objects. An object identity (oid) is associated with each entity. A real world object, irrespectively of its complexity, may be modeled as an occurrence of a single entity type: entity types may comprise any number of attributes which may in turn, iteratively, consist of other attributes. The structure of an entity type can be regarded as a tree whose root and inner nodes are respectively represented by the entity type and its composing attributes, as shown in figure 3.1.a. Attributes, entities, and relationships may be valued in multisets (i.e. allowing duplicates).

ERC+ fully supports the concept of relationship type. Several (two or more) entity types may be linked by a relationship type. As entity types, a relationship type may have attributes. Relationship types may be cyclic (a relationship type may bind twice -or several times- the same entity type, each time with a different role).

A problem is: do relationships have oids? Relationship types are not full objects like entities, therefore one could say that a relationship does not deserve an oid. Moreover, unlike entity types, relationship types do not form a generalization hierarchy. But there are two reasons for describing relationships with oids:

1/ Duplicate relationships can coexist, linking the same entities. For example, in the vineyard data base of figure 3.1 b, if a wine grower harvests two times the same year the same vineyard, picking up the same quantity, two duplicate relationships harvest will coexist. (Some very sunny years, a few grapes mature later and are harvested one month after the main harvest.)

2/ Updating a relationship type requires to be able to select its occurrences which are to be modified or deleted. Oids are then useful as handles to designate those occurrences.

In ERC+, relationships have oids which are different from those of entities.

Furthermore, ERC+ supports two other kinds of links between entity types: the “is-a” and the “may-be-a” link. Both are used to relate two entity types which to some extent represent the same real world objects, each one with a different point of view. Entities representing the same real world object share the same oid which is associated with the real world object.

- The classical "is-a" link (or generalization link) specifies that the population of an entity type ES is a sub-class (specialization) of another entity type EG (generalization). Thus, every oid contained in ES is also an oid of EG. This link is depicted in ERC+ diagrams by an arrow: ES $\rightarrow$ EG .
  An entity type may be a sub-class (super-class) of several generic (respectively, specific) entity types. For example, in a faculty database, one may want to describe the group of pd students who give courses as a sub-class of both the Faculty and the Student entity types. No cycle is allowed in the generalization graph. It is a lattice.

- The "may-be-a" link (or conjunction link) between two entity types E1 and E2 is used to specify that some (possibly all) entities of E1 describe the same real world objects as entities of E2 do, and vice-versa [Pernici 90, Richardson 91]; i.e. the populations of E1 and E2 may share common oids. The may-be-a link is graphically represented by a dashed line between E1 and E2: E1 - - - E2 .

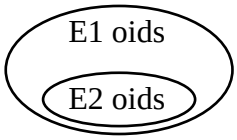
The is-a and may-be-a links allow to model the following three situations which may arise between the sets of oids of two entity types E1 and E2:

- the sets of oids are disjoint: there is no link in between E1 and E2.

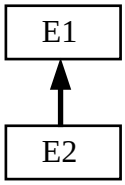
Venn diagram **ERC+ diagram**

- the set of oids of one entity type (E2) is included in the other set: this is an **is-a** link.

Venn diagram

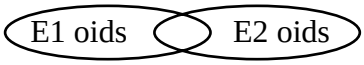


ERC+ diagram

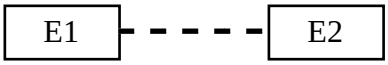


- E1 and E2 share some common oids: this is a **may-be-a** link.

Venn diagram



ERC+ diagram



Both is-a and may-be-a links allow inheritance of properties from one entity type to another one. The inheritance mechanism represents the means by which users can gather the information scattered over entity types that represent different points of view of the same real world object (i.e. entity types bound by is-a or may-be-a links). Inheritance can apply to all the properties attached to an entity: attributes, relationships, as well as is-a and may-be-a links. The algebraic operation (i-join) used to implement the inheritance mechanism is presented in section 5.

3.1. An example: the vineyard database

Figures 3.1 depict an example that will be used to highlight the important points of the formal description of the ERC+ model presented in sections 3, 4 and 5. It represents a vineyard database which comprises the following entity types:

- Vineyard: the Vineyard entity type (figure 3.1.a) includes a set of attributes to describe its label, the location of the vineyard, and the size of the area planted. It also includes a multivalued attribute "best years" to record good harvest years for each vineyard. Whenever a vineyard is planted, in total or in part, the data base records the year of the planting, the percentage of the vineyard that is planted and the plants (the plant brand identification and the number of plants planted). These informations together form a complex multivalued attribute “planting”. This attribute is multivalued, as the same vineyard may undergo different partial plantations over different years.

- Red Vineyard: this entity type is a sub-class of Vineyard. In addition to the attributes that it can inherit from Vineyard, it has a specific attribute to describe the exposure of each red vineyard to the sun light.

- White Vineyard: in addition to the inheritable attributes, White Vineyard includes a specific attribute "heating" to describe if there is any device in the vineyard to prevent spring frosts.

- Owner: it is an entity type used to represent vineyards owners. For each owner, its name is recorded.

- Wine Grower: for each wine grower the database stores its lastname, firstname, and address. A wine grower may also be an owner and vice versa. Thus, a may-be-a link is drawn between the entity types Owner and Wine Grower in the diagram.

The vineyard database also contains three relationship types: "Owns" which links the entity types Vineyard and Owner, "Harvest" which connects the entity type Wine Grower to Vineyard, and a cyclic relationship "Family" on the entity type Wine Grower. Note that the relationship type "Harvest" has two simple attributes to describe the year of a harvest and the quantity harvested.

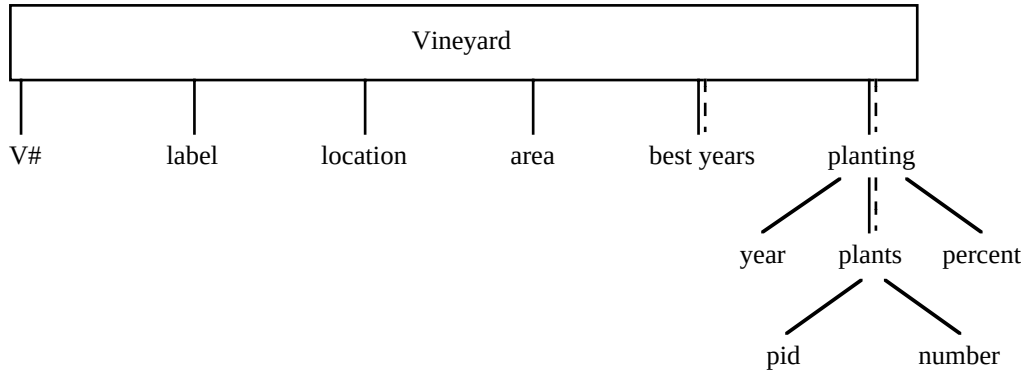


Fig. 3.1: (a) The Vineyard entity type

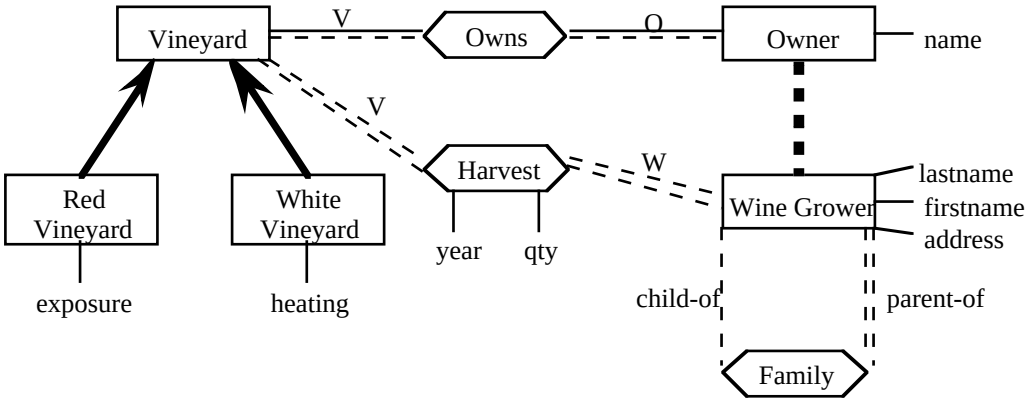


Fig. 3.1: (b) The ERC+ schema of the example vineyard database
(not showing Vineyard attributes)

The diagrams in figures 3.1 use the following graphical notations: a single straight line denotes a monovalued attribute or role, a double line (one straight, one dotted) denotes a multivalued mandatory attribute or role, a double dotted line denotes a multivalued optional attribute or role. V, O, W, child-of and parent-of are role names (see 3.2).

3.2. Formal definition of the ERC+ model

• Domains

Domains define the set of all possible values for an attribute, an entity or a relationship type. Values may be either atomic, like Mary or 1991, or complex, i.e. composed of other values. A complex value is a set of pairs <attribute name , v> where v is either a value or a multiset of values. For example, a complex value for a Vineyard entity may be:

{ (*V#*, 1335),
(*label*, Cote de Beaune),
(*location*, Bourgogne),
(*area*, 1000),
(*best_years*, {1986, 1988, 1989}),
(*planting*, { < (*year*, 1980),
(*plants*, { < (*pid*, pinot111), (*number*, 1000) >,
< (*pid*, pinot113) (*number*, 5000))> },
(*percent*, 100) } >) }

Let ED be the set of domains of atomic values, called elementary domains. These are defined as follows.

Elementary domain: ed

ed ∈ **ED** ⇔ ed=**(name,V,R)**

- name(ed) ∈ **NAMES** (where name(ed) is the name of the domain and **NAMES** is the set of all names)
- V(ed) is the set of elementary (atomic) values in the domain
- R(ed) is the set of relational operators defined on $(V(ed))^2$. ♦

Example : an elementary domain to be used for the attribute label above could be defined as:
(label_domain, {Pommard, Cote de Beaune, Macon, ...}, {=,≠}) .

Let **CD** be the set of complex domains. A complex domain is a set of complex values with the same format. There is also a special complex domain whose purpose is to represent the lack of value of relationships and entities which have no attribute. Complex domains are defined as follows.

Complex domain: cd

cd ∈ **CD** ⇔ cd=(name, V) such that:

name(cd)∈**NAMES** is the name of the complex domain
V(cd)={Ø} (the empty domain of entities and relationships without attributes)
or
Non empty complex domains are sets of complex values which all adhere to the same format. This format specifies, for each component of the complex value, its name, its domain and its cardinalities:

$\exists I=\{1,2,....,n\}$, $n\in \mathbb{N}^*$ $\exists F = \{ (A_i,d_i,min_i,max_i) / i\in I \}$, such that :

$\forall i\in I$, $A_i\in$ NAMES $\wedge$ $d_i\in$ (**ED** $\cup$ **CD**) $\wedge$ $min_i\in\mathbb{N} \wedge max_i\in\mathbb{N}^* \wedge min_i\leq max_i$

$\wedge \forall(k,j)\in \mathbb{N}^2$ $A_k=A_j \Rightarrow (A_k,d_k,min_k,max_k) = (A_j,d_j,min_j,max_j)$

and $V(cd)=\{ \{(A_i,\alpha_i)/i\in I\} / \forall i\in I, \exists(A_i,d_i,min_i,max_i)\in F \wedge \alpha_i\in P^{min_i:max_i}(V(d_i)) \wedge$

$\forall(k,j)\in \mathbb{N}^2$ $A_k=A_j \Rightarrow (A_k,\alpha_k)= (A_j,\alpha_j) \}$ ♦

where P(V) means the powerset of V, extended to multisets, called power-multiset of V;
and $P^{m:n}(V)$ means the power-multiset of V, generating multisets containing at least m elements and at most n elements (including duplicates).

Example: a complex domain to be used for a complex attribute plants (with two atomic components: pid and number) could be defined as:

(plants_domain , { { (*pid*, α_1), (*number*, α_2) } / $\alpha_1 \in$ V(pid_domain)
$\wedge \alpha_2 \in$ V(number_domain) }) .

A complex value in the above domain may be:
< (*pid*, pinot111), (*number*, 1000) >.

• Structures

The concept of structure bears the recursiveness necessary for complex object description. This concept conveys the characteristics of an attribute (its name and cardinalities, its composition and domain) independently from its association to the object it describes. This allows different attributes (representing similar properties for different object types) to share the same structure, which simplifies the formal definition of the algebraic operators, whose action often includes the creation of a new attribute with the same structure as the existing attribute it is derived from.

Let **S** be the set of structures. These are defined as follows.

S∈ **S** ⇔ S = **(name,min,max,comp,d)** such that:

- name(S) ∈ **NAMES** is the name of the structure (the attributes associated to the structure will have this name)

- $min(S)\in \mathbb{N}$, $max(S)\in \mathbb{N}^*$, $min(S)\leq max(S)$ are the minimum and maximum cardinalities of the structure. These numbers limit the number of values (including duplicates) the associated attributes may have in an instance of the object the attribute relates to.

If min(S)=0, S defines an optional attribute (with respect to the object the attribute relates to);
if min(S)≥1, S defines a mandatory attribute;
if max(S)=1, S defines a monovalued attribute;
if max(S)≥2, S defines a multivalued attribute.

- comp(S) = { S_i / $i \in I \wedge S_i \in \mathbf{S}$ } , I={1,2,....,n} , $n \in \mathbb{N}^*$ or I=Ø is the composition of the structure. If comp(S) = Ø the structure defines an atomic attribute. Otherwise, comp(S) is the set of the structures of the component attributes.
- d(S) is the underlying domain of the structure:

if comp(S)=Ø then $d(S) \in \mathbf{ED}$ (d(S) is an elementary domain)

else $d(S) \in \mathbf{CD}$ (d(S) is a complex domain)

The set of complex values of d(S) is a derived information:

$V(d(S)) = \{ \{(name(S_i), \alpha_i)/i \in I\} / \forall i \in I, \alpha_i \in \mathbf{P}^{min(Si):max(Si)}(V(d(S_i))) \}$ ♦

Example : the structure of the plants attribute in figure 3.1.a is defined as:
(plants,1,n, { (pid,1,1,Ø,pid_domain), (number,1,1,Ø,number_domain) }, plants_domain).

• Entity types

An entity type is defined by its name, its schema, which is the set of the structures of its attributes, its generic entity types, the set of the entity types with which it is in conjunction, and its population, which is a set of occurrences (entities) with their oids and values.
Let **E** be the set of entity types. These are defined as follows.

$E \in \mathbf{E} \quad \Leftrightarrow \quad E = \mathbf{(name, sch, gen, muid , pop)}$ such that:

- $name(E) \in \mathbf{NAMES}$ is the name of the entity type

- sch(E) = { S_i / $i \in I \wedge S_i \in \mathbf{S}$ } , I={1,2,....,n} , $n \in \mathbb{N}^*$ or I=Ø is the schema of the entity type. It is a (possibly empty) set of structures.

- gen(E) = { (EG_j) / $EG_j \in \mathbf{E}$ } , is the possibly empty set of the generic entity types of E.

Let EG be any generic entity type of E, then to each occurrence of E corresponds an occurrence of EG which describes the same real world object:

$soid(E) \subseteq soid(EG)$

must be satisfied at any time; soid(E) is the set of oids of E and is formally defined below.

- muid(E) = { (E_j) / $E_j \in \mathbf{E}$ } , is the possibly empty set of the entity types bound to E by conjunction.

- pop(E) = {(oid, val)/ $val \in d(E)$} is the population of the entity type. It is a set of entities. Each entity is a couple made up of the entity identity (oid) and its value. The value is an element of the domain of the entity type, d(E), which is a derived information:

$d(E) \in \mathbf{CD}$

$V(d(E)) = \{\{(name(S_i), \alpha_i)/i \in I\} / \forall i \in I, S_i \in sch(E) \wedge \alpha_i \in \mathbf{P}^{mi_i: ma_i}(V(d(S_i))) \}$

where: $mi_i=min(S_i)$ and $ma_i=max(S_i)$

The set of oids of E is called soid(E), and is formally defined by:

$soid(E) = \{e / \exists v, (e,v) \in pop(E) \}$ ♦

Example: the definition of the entity type White Vineyard of figure 3.1.b is:
(White_Vineyard, { (heating, 1, 1, Ø, heating_domain) }, {Vineyard}, Ø, pop)
where pop describes the White Vineyard entities stored in the database.

• Relationships types

A relationship type is defined by its name, the set of entity types it links, with the description of the characteristics of the links (role names and cardinalities), the set of structures of its attributes (which constitutes its schema) and the set of its occurrences.

Let **R** be the set of relationship types. These are defined as follows.

$R \in \mathbf{R} \quad \Leftrightarrow \quad R = \mathbf{(\ name, pet, sch, pop\ )}$ such that:

- $name(R) \in \mathbf{NAMES}$ is the name of the relationship type
- $pet(R)$ is the set of entity types participating in the relationship type. For each entity type, its role and the minimum and maximum cardinalities of its link to the relationship type are specified:

$pet(R) = \{ (E_j, role_j, min_j, max_j) / j \in J \wedge E_j \in \mathbf{E} \wedge role_j \in \mathbf{NAMES} \wedge min_j \in \mathbb{N} \wedge max_j \in \mathbb{N}^* \wedge min_j \leq max_j \}, \quad J=\{1,2,....,p\}\ ,\ p \in \mathbb{N}^*\ ,\ p>1,$

within the relationship type, the role names are unique:

$\forall\ ((E_1, role_1, min_1, max_1), (E_2, role_2, min_2, max_2)) \in (pet(R))^2\ ,$

$role_1 = role_2 \quad \Rightarrow \quad (E_1, role_1, min_1, max_1) = (E_2, role_2, min_2, max_2)$

- $sch(R) = \{ S_i / i \in I \wedge S_i \in \mathbf{S} \}\ ,\ I=\{1,2,....,n\}\ ,\ n \in \mathbb{N}^*$ or $I=\emptyset$ is the schema of the relationship type. It is the (possibly empty) set of the structures of its attributes.
- $pop(R) = \{(oid, poc, val)\}$ is the population of the relationship type. It is a set of relationships. Each relationship is a tuple made up of the relationship identity (oid), the set of the linked entities (poc) and the relationship value (val).

$\forall r \in R,\ poc(r) = \{ \{(E_j, role_j, e_j) / (E_j, role_j, min_j, max_j) \in pet(R)\} / \forall j \in J,\ e_j \in soid(E_j) \}$

$\forall j \in J,\ \forall e_j \in soid(E_j),\ min_j \leq card(\{ r / r \in pop(R) \wedge (E_j, role_j, e_j) \in poc(R)(r) \}) \leq max_j$

The value of a relationship is an element of the domain of the relationship type, $d(R)$, which is a derived information:

$d(R) \in \mathbf{CD}$

$V(d(R)) = \{ \{ (name(S_i), \alpha_i) / i \in I \} / \forall i \in I,\ S_i \in sch(R) \wedge \alpha_i \in \mathbf{P}^{mi_i} : ma_i(V(d(S_i))) \}\ ,$

where : $mi_i = min(S_i)$ and $ma_i = max(S_i)$

The set of oids of R is called soid(R), and is formally defined by:

$soid(R) = \{e / \exists v, \exists role, (e, role, v) \in pop(R) \}$ ♦

Example: the relationship type Harvest of figure 3.1.b is defined by:
(Harvest, { (Vineyard, V, 0, n) , (Wine_Grower, W, 0, n) }, { (year, 1, 1, Ø, year_domain), (qty, 1, 1, Ø, qty_domain} , pop)
where pop describes the set of occurrences stored in the database.

• Attributes

An attribute is defined by the object to which it is attached, its structure and its values for each occurrence of its object.
Let **A** be the set of attributes. These are defined as follows.

$A \in \mathbf{A} \quad \Leftrightarrow \quad A = \mathbf{(\ obj, str, inst\ )}$ such that:

- $obj(A) \in (\mathbf{E} \cup \mathbf{R} \cup \mathbf{A})$ is the object (entity type, relationship type or complex attribute) to which the attribute is attached.
- $str(A) = (name(A), min(A), max(A), comp(A), d(A))$ is the structure associated to the attribute

$str(A) \in \mathbf{S}$

if $(obj(A) = E_i \wedge E_i \in \mathbf{E})$ then : $str(A) \in sch(E_i)$

if $(obj(A) = R_j \wedge R_j \in \mathbf{R})$ then : $str(A) \in sch(R_j)$

if $(obj(A) = A_k \wedge A_k \in \mathbf{A})$ then : $str(A) \in comp(A_k)$

- inst(A) is the instanciation of the attribute. It is a total function associating to each value of the object to which the attribute is attached, the component value (which may be a multiset) of the attribute.
inst(A) is defined by:

$$\text{inst(A): } V(d(obj(A))) \rightarrow \mathbf{P}^{\min(A):\, \max(A)}(V(d(A))) \text{ , such that :}$$

$$\forall\, v \in V(d(obj(A))) \text{ , } \quad inst(A)(v) = proj_{name(A)}(v) \text{ .} \qquad \blacklozenge$$

• Identity axioms

Let gen*(E) be the set of ancestors of an entity type E in the generalization graph.
Let gen+-(E) be the set of entity types to which an entity type E is connected through a path of the generalization graph.

A1. There is no cycle in the generalization graph:

$$\forall E \in \mathbf{E} \quad E \notin gen^*(E) \qquad \blacklozenge$$

A2. Two entity types linked by a path in the generalization graph can share oids. Defining a conjunction link in between is useless:

$$\forall E1 \in gen+\text{-}(E2) \quad E1 \notin muid(E2) \qquad \blacklozenge$$

A3. Two entity types, E1 and E2, which are bound neither by a generalization link nor by a conjunction link, cannot have any common oid:

$$\forall E1 \in \mathbf{E} \quad \forall E2 \in \mathbf{E} \quad (E1 \notin gen+\text{-}(E2) \wedge E1 \notin muid(E2)) \quad \Rightarrow \quad soid(E1) \cap soid(E2) = \emptyset \qquad \blacklozenge$$

4. Objects manipulation languages

The previous section has presented a formal description of the concepts used by the ERC+ model. Most of actual OO data models, extended ER models and OR models share the same new concepts: object identity, generalization, object with complex structure, composition link and/or more general relationship link between objects. In this section, we discuss the guidelines and requirements for data manipulation languages for managing these new concepts; in the following section we describe more precisely the operators of the ERC+ algebra.

One of the major principles of algebra's is the closure property which requires the result of any operation to be of the same type as the operands. This allows to build expressions involving nested operators. In OO models, where the object class is the main concept, algebraic operators are performed on classes of objects and the result of any operation is a class of objects. In extended ER (or OR) models, there are several basic concepts: entity type, relationship type, attribute or value. Entity types are the main kind of objects, thus languages provide operators for manipulating them. In order to keep extended ER languages from being very complex, operations for manipulating relationships are typically not provided by these languages. Instead, attributes and relationships are queried through the entities to which they are bound.

Each new concept generates in the data manipulation languages either new capabilities (for instance, the crossing of reference attributes or of relationships) or modifications of the relational operators (for instance, the union of objects is based upon their oids). Those novelties are discussed hereinafter.

• Oids

The inclusion of oids in data models requires the addition of a new comparison operation and the modification of the usual set operations to handle objects with identities. Firstly, in order to be able to express cyclic queries involving twice the same object, a new equality operator is defined. For example, in a chess tournament one could want to verify if there is no error in the planning by asking: "Is there a player who is scheduled to play against him/her-self?"

OO DMLs must provide two equality comparison operators:
- o1==o2 which is true when o1 and o2 are the same objects (same oid) irrespectively of their values,
- o1=o2 which is true when the values of o1 and o2 are the same.

Secondly, set operators (union, difference, intersection) of the relational model need to be modified, as their original definition is value oriented. In the relational data model, a relation can not contain two identical tuples. In OO data models, each object has a specific identity and thus two objects having the same value can coexist in the same class. OO set operators must therefore be based on the comparison of the oids and not on the comparison of values. To illustrate this, consider the following example:

- Union in the relational data model:

R1: {v1, v2, v3}
R2: {v1, v2, v4}

$R1 \cup R2 = \{v1, v2, v3, v4\}$

- Union in data models with oids:

E1: {<o1, v1>, <o2, v2>, <o3, v3> }
E2: {<o1, v1>, <o2', v2>, <o4, v4>}
where **<oi, vi>** represents an object: **oi** is its oid and **vi** its value.

$E1 \cup E2 = \{<o1, v1>, <o2, v2>, <o3, v3>, <o2', v2>, <o4, v4>\}$

Using comparison operations based on oids, the union operation may be performed on two classes containing objects that have the same identity but different values. What will be the values of the resulting objects in the following example?

E1: {<o1, v1>, <o3, v3>}
E2: {<o1, v2>, <o4, v4>}

$E1 \cup E2 : \{<o1, f(v1,v2)>, <o3, v3>, <o4, v4>\}$

The objects identified by o1 above represent two different views of the same real world object. For instance, in a university database, E1 and E2 may represent a Faculty entity type and a Student entity type, respectively. o1 represents both a faculty and a student (a PhD student who teaches a course). The value corresponding to the resulting object o1 in the union of E1 and E2 is neither v1 nor v2. It is instead composed of both v1 and v2. How this resulting value is composed depends upon the data manipulation language.

Lastly, with the relational model, relations used as operands of set operators must have the same type (same set of attributes) to allow the comparison of their values. With OO models, on the contrary, set operations are allowed to have operands with different associated types. Existing OO DMLs choose different solutions for defining the type associated with the result of a union: it can be made of either the common attributes of the operands (or the nearest common ancestor in the generalization hierarchy) [Loizou 91, Vrbsky 89] or the union of all their attributes. In ERC+ the later solution has been adopted because it provides more descriptive information (see the discussion below about placing the result into the generalization graph).

• Generalization and conjunction

The inclusion of the concepts of generalization and/or conjunction in a data model generates two new aspects that must be taken into account by DMLs associated with the model: inheritance of class properties and placement of the result of a query in the generalization graph.

Inheritance applies to attributes and methods as well as to links between object classes. These links are: reference links (or composition links) in OO models and relationship, generalization and conjunction links in extended ER models. Inheritance through generalization links is usually descending (downward from the super-class to the sub-class): each object of a sub-class is also an

object of the super-class and inherits the properties associated with the objects of the super-class. But ascending inheritance can also be useful. It is a mechanism by which the properties of the sub-class object are attached to the corresponding object (having the same oid) in the super-class. For example, in the vineyard database one can wish to produce a list of all the vineyards by listing for each vineyard its properties together with the exposure attribute if it is a red vineyard, or with the heating attribute if it is a white one.

With explicit inheritance, whenever a user wants to use an attribute or a property of a super-class or a sub-class, (s)he must make an explicit reference to the name of the class to which the attribute belongs. By contrast, with implicit inheritance users can use properties of the super- or sub-class and let the system fill in the references of the corresponding classes. Most user friendly DMLs such as SQL-like and graphical languages support implicit descending inheritance. This lightens the burden of writing queries: users can quote any property of any super-class, the system will look for it upward in the generalization hierarchy.

In order to get complete DMLs (with descending and ascending inheritance), the ERC+ algebra offers descending and ascending inheritance through is-a links. May-be-a links are not oriented, thus inheritance is also possible in both directions. In order to solve the ambiguity which may arise when an entity type has several generic entity types which have properties with the same name, inheritance in the ERC+ algebra is explicit: an identity-join operator allows users to join two entity types connected by a generalization or conjunction link.

The result of a query is an object class; thus it has to be placed in the generalization graph in order to express its generalization/conjunction relationships with the other object classes. There are two main solutions:
1. the result of a query describes new real world objects. It is a class containing new oids, placed at the top of the hierarchy;
2. the result of a query is another point of view on real world objects which are already described (with other points of view) in the database. It is a class containing existing oids, i.e. a sub-class (or a derived class) of some existing class(es).

DMLs of the first type are called object generating languages. The properties of the new class are derived from the properties of the operands [Bancilhon 88]. In OO models, this can be achieved by defining reference attributes which point at the operand objects [Bertino 92]. DMLs of the second type are called object preserving languages. The properties of the new class are derived from the operand through the sub-class mechanism [Scholl 90, Sunye 92]. The object preserving solution generates graphs which are clearer, more easily understandable by users. ERC+ DMLs are object preserving. The result of a query, however complicated, involving one or several entity types, is an entity type which is derived from the operand entity types and whose occurrences are derived from those of the operand(s): same oids, derived values.

• Complex types

A large amount of research effort has been devoted to DMLs for models with complex types such as non first normal form relational models [Jaeschke 82, Roth 84] and complex objects models [Carey 88, Bancilhon 88, Zaniolo 83]. OO models and extended ER models often support, as ERC+ does, complex attributes, which are composed of other component attributes, and multivalued attributes which can have several values. A multivalued attribute may be set, multiset, list, or array-valued. An example of complex and multivalued attribute is the attribute planting in figure 3.1.a. Complex and multivalued attributes may include multivalued attributes at several levels of nesting, as the attribute plants in planting.

With complex types, the DML must provide tools for working on a value inside a set (or multiset, list ...) of values and at any depth in the complex type. Usual tools are:

1/ set operators ($\in$, $\subseteq$, $=$),

2/ attribute-variables at each level of multivaluation, associated with a quantifier ($\exists$ or $\forall$),

3/ nested queries, one query for each level of multivaluation.

Nesting of queries is a commonly used solution in NF2 relational models where a multivalued attribute can take a nested relation as value [Jaeschke 82]. The operators that are used to manipulate relations are also used to work on these relation-valued attributes. By contrast, in languages with two different concepts, object (or entity) and attribute, different operators must be defined to manipulate the different concepts. For example, in the ERC+ algebra the selection operator is defined on entity types, and a new operator (the reduction) applies to a multivalued attribute of an entity type, and is used to eliminate in each occurrence of the entity type, inside the multivalued attribute, the values which do not satisfy a given predicate.

Projection is another operator which works on a complex structure. In NF2 relational models, users have to write nested projections when they want to prune a complex attribute. In models using several concepts (as ERC+), the projection operator is defined on the entity type and a dot notation allows to define the pruning of component attributes at any level.

• Relationships and reference attributes

Relationships in ER models and reference attributes in OO models are used to link objects together. The associated DML should allow users to select objects based on their links or relationships to other objects. For instance, in the vineyard database, a user may want to list all the growers which harvested "Pommard" wine in 1990. Most OO SQL-like languages allow access from one object to related objects through reference links. They often use a dot notation to move across reference and value attribute links.

An alternative solution, better suited to algebra's, is to respect the following principle: each operator manipulates only its operands. Objects which are not explicitly stated as operands of an operator cannot be accessed by it. ERC+ algebra abides by this rule. A query which involves more than one entity type must be formulated as an expression where operators which gather together into a new complex entity type several entity types linked by a relationship (r-join) or by an is-a or may-be-a link (i-join), are used at first.

5. The ERC+ algebra

This section presents the query operators associated with the ERC+ model. The ERC+ algebra is a set of primitive operators which may be combined in any order into expressions, so that any possible user query on an ERC+ database can be satisfied by an appropriate algebraic expression.

The closure property and the power of the algebra rely on five basic assumptions:
> operands and result are entity types.
Each operator is designed to manipulate one (or more) entity type(s) and to build the result as a derived entity type. Thus, the result may serve as an operand for a subsequent operator.

> resulting entity types are complemented with relationship types, generalization and conjunction links derived from the ones linking the operands. This allows to (temporarily) integrate computed entity types into the existing database, so that they may be used in further manipulations, just as normal entity types. Algebraic expressions of any complexity may thus be defined. The ERC+ algebra is closed.

> the algebra is object preserving.
Each occurrence of the result is derived from the operand entities in the following fashion. It is made up of:
- its oid which is identical to the oid of the operand entity from which it comes. For instance, the oids in the result of the following selection:
select [firstname="Paul"] WineGrower
are the oids of the Paul wine grower(s);
- its value which is derived from the values of the operand(s). For instance, the value of the above selection is the value of the corresponding wine grower. The value of an occurrence of the following projection:
project [lastname] WineGrower

is the value of the corresponding wine grower, restricted to the lastname attribute;
- its relationships which are derived as equal to those binding the operand;
- its links of generalization and conjunction which are derived as equal to those binding the operand.

> the algebra is consistent with the model.

As the model supports complex objects, the operators build their resulting entity types as complex objects. In particular, the product, relationship-join and identity-join operators (used to collapse different entity types into a single object) use complex attribute structures to insert the information from the other operands into the "main" (the first) operand entity type. Product and relationship-join are non flat operators: they create one occurrence for each occurrence of the main operand. The occurrences of the other operand(s) are collapsed into one complex multivalued attribute. For instance, the relationship-join of Owner and Vineyard through the relationship Owns:

O := Owner **r-join** (Owns, Vineyard)

creates a new entity type O whose schema comprises the schema of Owner augmented by a complex attribute, grouping all the attributes of Vineyard.

This particularity of the ERC+ algebra has two main advantages:
- it fulfills users' requirements to produce non flat results or forms that show for each occurrence of the main entity all the second entity occurrences bound to it;
- binary operators with a main operand (product and relationship join) are object preserving: the oids of the result are derived from (are equal to) those of the main operand. With flat operators (as in the relational algebra), this would have been impossible.

> inheritance through is-a and may-be-a links is explicit and two ways.

The algebra offers an operator, the identity-join, which allows users to group two entity types linked by either an is-a or a may-be-a link into a single entity type (with all the attributes, relationships, generalization and conjunction links of both entity types). All the other operators use only the own properties (not inherited) of their operands.

The algebra includes 9 primitive operators. Derived operators may also be defined in order to lighten the writing of queries involving several operators. In the following, we briefly present the operators.

• The **selection** operator is similar to the relational one. The operation:

E = **select** [predicate] E1,
where E1 is an entity type,

creates a derived entity type E whose population is the subset of the population of E1 for which the predicate is true. The schema of E, the relationships, generalizations and conjunctions linking E are derived from (equal to) those of E1.
The predicate may involve any attribute or component attribute of E. For each multivalued attribute, a variable associated with a quantifier ($\exists$ or $\forall$) must be defined.

Example:
List the vineyards which had a best year in 1983 and one in 1990.

select [$\exists$b1$\in$ bestyears, $\exists$b2$\in$ bestyears (b1=1990 $\wedge$ b2=1983)] Vineyard

• The **projection** operator is similar to the relational one. The operation:

E = **project** [attribute-list] E1,
where E1 is an entity type,
and attribute-list is a list of attributes of E1 and/or sub-structures of attributes of E1,

creates a derived entity type E whose oids are the same as those of E1. The relationships, generalizations and conjunctions linking E are derived from (equal to) those of E1. The schema of E is made up of the attributes or sub-attributes as defined in the attribute list. The sub-structures of the attribute list are defined by using the usual dot notation. The value of each E occurrence is derived from the value of the corresponding E1 occurrence, by pruning it according to the attribute list.

Example:
For each vineyard list its label and its planting years.

project [label, planting•year] Vineyard

• The **reduction** operator is a new operator. It complements the functionality's offered by the selection and projection operators with respect to the goal of selecting the desired information from an entity type. While selection and projection allow users to discard occurrences or attributes not of interest to them, reduction allows the elimination of attribute values which do not conform to a given predicate.
The operation:

E = **reduce** [A / predicate] E1
where E1 is an entity type,
and A is a (direct or component) attribute of E1,

creates a derived entity type E whose oids are the same as those of E1. The schema of E, the relationships, generalizations and conjunctions linking E are derived from (equal to) those of E1.
The predicate must involve A or its component attributes. It may also involve other attributes of E1. It is defined as a selection predicate. The value of each E occurrence is derived from the value of the corresponding E1 occurrence, by retaining in A only the values which satisfy the predicate; the value of the other attributes is not modified.

Example:
List vineyards, showing in the vineyard best years only the years before 1960.

reduce [bestyears / bestyears<1960] Vineyard

• The **union** operator is different from the relational one, which is value based. It merges the populations of two entity types according to their oids. It is a binary operator with two main operands. The environment of the result (relationships, generalizations and conjunctions) is derived from both operands. The operation:

E = E1 **union** E2
where E1 and E2 are two entity types,

creates a derived entity type E whose oids are the same as those of E1 plus those of E2. The schema of E is derived by union-fusion of E1 and E2 schemas. Each attribute, Ai, of E1 (and of E2) is also an attribute of E. If E2 has no attribute of the same name, the Ai attribute of E is identical to the Ai attribute of E1 (the only difference is that Ai becomes optional). If E2 has also an Ai attribute, the Ai attribute of E is the union of both attributes: its domain is the union of both domains, its value is the multiset-union of the two values. In the same way, the relationships, generalizations and conjunctions linking E are derived from those of E1 and those of E2 by union-fusion (fusion of those with identical names and linking the same entity types).

Example: let us suppose that the vineyard database above contains, in addition to red and white vineyards, also green, gray, and other vineyards.
List the red and white vineyards.

RedVineyard **union** WhiteVineyard

The result of this query is a sub-class of Vineyard, which is implicitly linked by a conjunction link with White Vineyard and Red Vineyard, and which has two optional attributes, exposure and heating.

• The **r-join** operator (for relationship-join) is a n-ary operator with one main operand. It is used to transform a network of entity types into a hierarchical structure (a single entity type). In some sense, this operator groups into a single entity the information scattered over entities linked by a relationship. From an object-oriented point of view, a r-join may recompose as a single object a complex object which has been disassembled into its component objects.
The operation:

E = E1•role1 **r-join** (A : R, E2•role2 , ... , En•rolen)
where rolei is the role played by the Ei entity type in the relationship type,
(the specification of the roles is mandatory only in a cyclic relationship)
and A is a name for the new complex attribute that will be created,

creates a derived entity type E whose oids are the same as those of E1 (E1 is the main operand). The relationships, generalizations and conjunctions linking E are derived from (equal to) those of E1. The schema of E is made up of the attributes of E1 plus a new complex multivalued attribute named A which is derived from the schemas of E2, ... En and R. The value of an occurrence of E is made up of the value of the corresponding E1 occurrence plus the multiset of the E2, ... En and R occurrence values which are linked to E1 (if there is some).

Example:
List each wine grower with its vineyards.

WineGrower **r-join** (V : Harvest , Vineyard)

The result contains one occurrence for each wine grower.

• The **i-join** operator (for identity-join) is a binary operator with one main operand. It is used to group into a single entity the information scattered over two entities bound by a generalization or a conjunction link. It allows the user to merge two points of view on the same real word objects.

The operation:

E = E1 **i-join** (A : E2)

where E1 and E2 are two entity types bound by a is-a or a may-be-a link,

and A is a name for the new complex attribute that will be created

creates a derived entity type E whose oids are the same as those of E1. The relationships, generalizations and conjunctions linking E are derived from (equal to) those of E1. The schema of E is made up of the attributes of E1 plus a new complex monovalued attribute, named A, which is derived from the schema of E2: it groups all the attributes of E2. The value of an occurrence of E is made up of the value of the corresponding E1 occurrence plus the value of the E2 occurrence (if it exists).

Example:
List all the information about the red vineyards.

RedVineyard **i-join** (V:Vineyard)

The result contains one occurrence for each red vineyard, with the exposure attribute plus all the attributes of Vineyard. The opposite i-join:

Vineyard **i-join** (R:RedVineyard)

would contain one occurrence for each vineyard, either red or not (in this case the exposure attribute in the result is optional).

• The **product** operator is used to collapse unrelated entity types into a single entity type. This is similar to a nested NF2 relational product, as each entity from the first operand is associated with all entities from the second operand. It is a binary operator with one main operand (the first one). The product is necessary to allow users to dynamically establish unexpected links (not expressed by relationship types in the schema) between unrelated entity types.

The operation:

E = E1 **product** (A: E2)

where E1 and E2 are two entity types,

and A is a name for the new complex attribute that will be created

creates a derived entity type E whose oids are the same as those of E1 (E1 is the main operand). The relationships, generalizations and conjunctions linking E are derived from (equal to) those of E1. The schema of E is made up of the attributes of E1 plus a new complex multivalued attribute named A which is derived from the schema of E2. The value of an occurrence of E is made up of the value of the corresponding E1 occurrence plus the multiset of all the E2 occurrence values.

The ERC+ algebra contains also two syntactic operators, renaming and simplification, which allow to conform the schema of an entity type to the rules of the model or of the algebra:

- the **renaming** operator changes the name of an attribute, to prepare the fusion of similar attributes during a union.
- the **simplification** operator deletes unnecessary complexity in the structures which may be built by other operators, projection in particular. Simply stated, simplification deletes one level in a complex structure whenever a complex attribute has only one component attribute (unless both are multivalued).

Derived operators may be defined in order to help users to write shorter queries.

A **difference** operator may be defined as derived through the appropriate composition of a product, a selection and a projection. An **intersection** may also be defined as two differences. As usual, the semantics of these operators is to form a new population corresponding to those occurrences from the first operand for which there is no occurrence in the second operand with the same oid (difference), or to those occurrences from the first operand for which there is an occurrence in the second operand with the same oid (intersection). Those two operators have a main operand, the first one.

The r-join is equivalent to the outer join of the relational model: there is one occurrence in the result for each occurrence of the main operand even if it is not linked by the relationship. A useful derived operator is the **sel-r-join** which deletes from the result all the occurrences which are not linked by the relationship. The sel-r-join is equivalent to an ERC+ expression made up of a r-join followed by a selection.
Example:
List the wine growers who are children of a family of wine growers:
WineGrower/child-of **sel-r-join** (F : Family, WineGrower/parent-of)

In the same way, another derived operator, **v-join** (for value-join), is a NF2 theta-join: it groups into a single entity type the information scattered over two entity types linked by a predicate on their attributes. It is equivalent to an ERC+ expression made up of a product followed by a reduction and a selection.
Example:
List the wine growers who have the same name as a wine label
WineGrower **v-join[**name=label**]** Vineyard
The result contains one occurrence for each wine grower who satisfies the condition.

ERC+ incorporates two basic languages: the algebra which was presented above, and a calculus [Parent 90]. These basic languages are the formal foundations on which user friendly languages such as SQL-like and graphical languages are built. An ERC+ SQL has been defined [Sunye 92], and an ERC+ graphical interface, which will be presented in the following sections, is being implemented.

6. Schema editor

Now that we have discussed the requirements of database applications and shown how the ERC+ model can be used to meet them, we present the SUPER design environment which is based on the ERC+ model. SUPER is an integrated environment of tools which aims at the specification and development of a consistent set of visual user interfaces for database design and manipulation. The tools in the current version of the prototype consist of a graphical schema editor, a data browser and an editor for the graphical specification of database queries and updates. The schema editor is presented in this section while section 7 presents the query editor. Figure 6.1 depicts an ERC+ schema diagram which will be used as a running example throughout the discussion of the graphical editors.

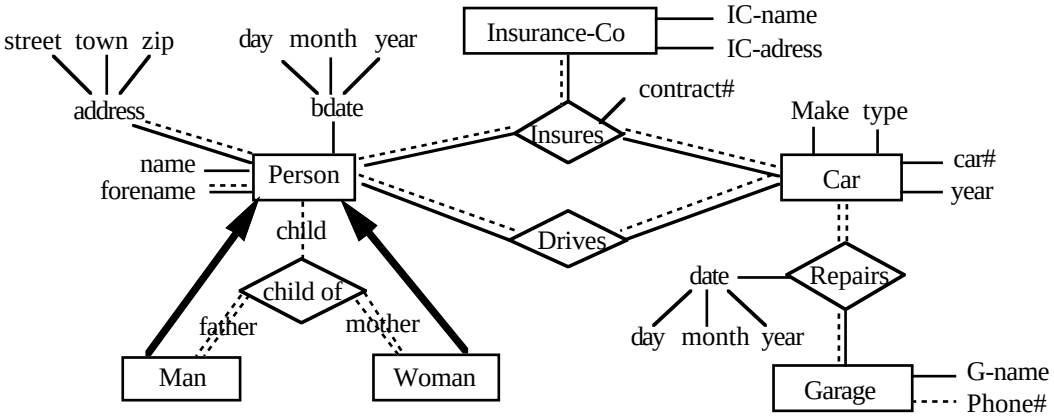


Figure 6.1: a sample ERC+ schema

The schema editor is a visual data definition interface, providing two modes for schema definition. Each mode has a separate display window, identified by the name of the schema being edited and labeled by the corresponding operation mode. In the **graphical** mode, the designer builds an ERC+ diagram by direct manipulation. The user picks the graphical symbols from a palette and positions them into the workspace provided in the associated window. The symbols in the palette correspond to ERC+ constructs (entity, relationship, attribute, generalization, conjunction). In the **alphanumeric**

mode, forms-like representations of ERC+ constructs (called **object boxes**) are provided by the editor for entering data definitions. The different object boxes correspond to the ERC+ constructs. Some are shown in figure 6.2.

Figure 6.2 shows an example of the graphical and alphanumeric windows during schema design. A user working simultaneously on several schemas is provided with both a graphical and an alphanumeric window for each schema. Standard editing operations are available through pull-down menus. "Schema", "Edit" and "Dictionaries" menus are available in both windows, and provide the same functionality's. The "Schema" menu contains the usual operations for opening, saving, creating, ... a schema. The "Edit" menu offers cut, copy and paste facilities, as well as undo and redo. The "Dictionaries" menu gives access to a global dictionary or any of the specialized dictionaries (entities, relationships, attributes).
The "Options" menu in the graphical window contains purely graphical manipulations (changing the layout, rearrange object disposal, ...) and is therefore specific to this window. Conversely, functionality's for creating a new schema (or modifying an existing one) are provided in the "Creation" menu when in the alphanumeric mode. They are equivalent to the definition of schema elements through the graphical palette.

As schema editing is a well known process, we limit the discussion on a few comments on three aspects: displaying, editing and browsing.

6.1 Information display

The graphical representation follows ERC+ guidelines. In a later version of the prototype users will have the possibility to customize the representation by choosing their own symbols. For user defined diagrams, SUPER stores the associated spatial information, so that the diagram may be later displayed as it was at creation time. For schemas defined in the alphanumeric mode, SUPER automatically builds and displays the corresponding diagram. The management of this process is a complex task, so we did not try to implement a sophisticated algorithm, leaving to users to adjust the diagram (dragging elements around) if they dislike it. For readability, attributes may be hidden.

Figure 6.2 shows the schema diagram for a hypothetical Insurance application. Each object in the diagram has been created by first selecting the corresponding graphical symbol in the palette and then clicking in the workspace to position the object. The creation of an object activates the display of the corresponding object box (alternatively, it may be displayed using the alphanumeric Creation menu). Newly created objects receive a standard name, which can be changed in the corresponding alphanumeric object box. An object box contains text entry areas (e.g. object's name and comment), radio buttons for predefined choices (cardinality specification, for instance) and list-bars referring to objects directly attached to the current object. The entity box (Person) in figure 6.2 shows list-bars for attributes, links and generalizations defined on an entity type. A list-bar for components of a complex attribute is included in the attribute box (address).

List-bars have been chosen as a standard technique to link objects. Clicking on a list-bar displays the corresponding scrollable list of attached objects. Two such lists are shown in figure 6.2, one for links on the Person entity type, one for components of the address attribute. Lists have a standard behavior. They group objects of the same type, attached to the same parent element (the latter is the schema for dictionary lists). Clicking on an object in a list displays its object box. Clicking on the New button in the list box displays an empty object box for adding a new object to the list. Using object boxes, list-bars and the attached lists, users may navigate through the schema and add or modify objects as needed. Top-down definition strategies are very easily performed.

Flexibility in the schema design process is enhanced by the possibility to leave object definitions incomplete. Users may, for instance, define entity types and relationship types, and come back later to these objects to attach attributes or add generalizations. Each user may follow his/her own strategy. Incomplete schema definitions may be saved and reused in another session. At any time, a validation function may be activated to check whether the actual schema definition is consistent with model rules. If inconsistencies or incompleteness are detected, they are reported to the user. As the editor keeps track of what has been validated (and is still valid), users may easily identify which definitions have to be refined to correct their schema.

Some model rules have to be permanently enforced (uniqueness of entity names, for instance) in order to avoid ambiguities. These rules are immediately checked by the editor. Default names are, of course, always generated unique.

Finally, users may quit the editor any time. When they come back to continue schema editing, their work on the schema is reactivated in exactly the same status as at the time of the interruption (open windows and object boxes, selected objects, ...).

6.2 Flexibility, reusability and backtracking

Besides the above usual editing functions, SUPER includes the additional facilities of redundancy, reusability and backtracking to make users' task as easy as possible.

Redundancy is intended to provide flexibility. It has already been introduced by allowing users to view their schema through two equivalent representations. Accordingly, some functionality's have been implemented redundantly, so that users may access them directly through the representation they are using. For instance, creating an object may be done graphically (with only one way to do it), or in several alternative ways in the alphanumeric mode. A new attribute, for instance, can be created either by activating the Creation menu, or clicking on the New button in the attribute list attached to its parent object. Whichever way is used, it will result in displaying an attribute creation box, where users will enter the attribute definition.

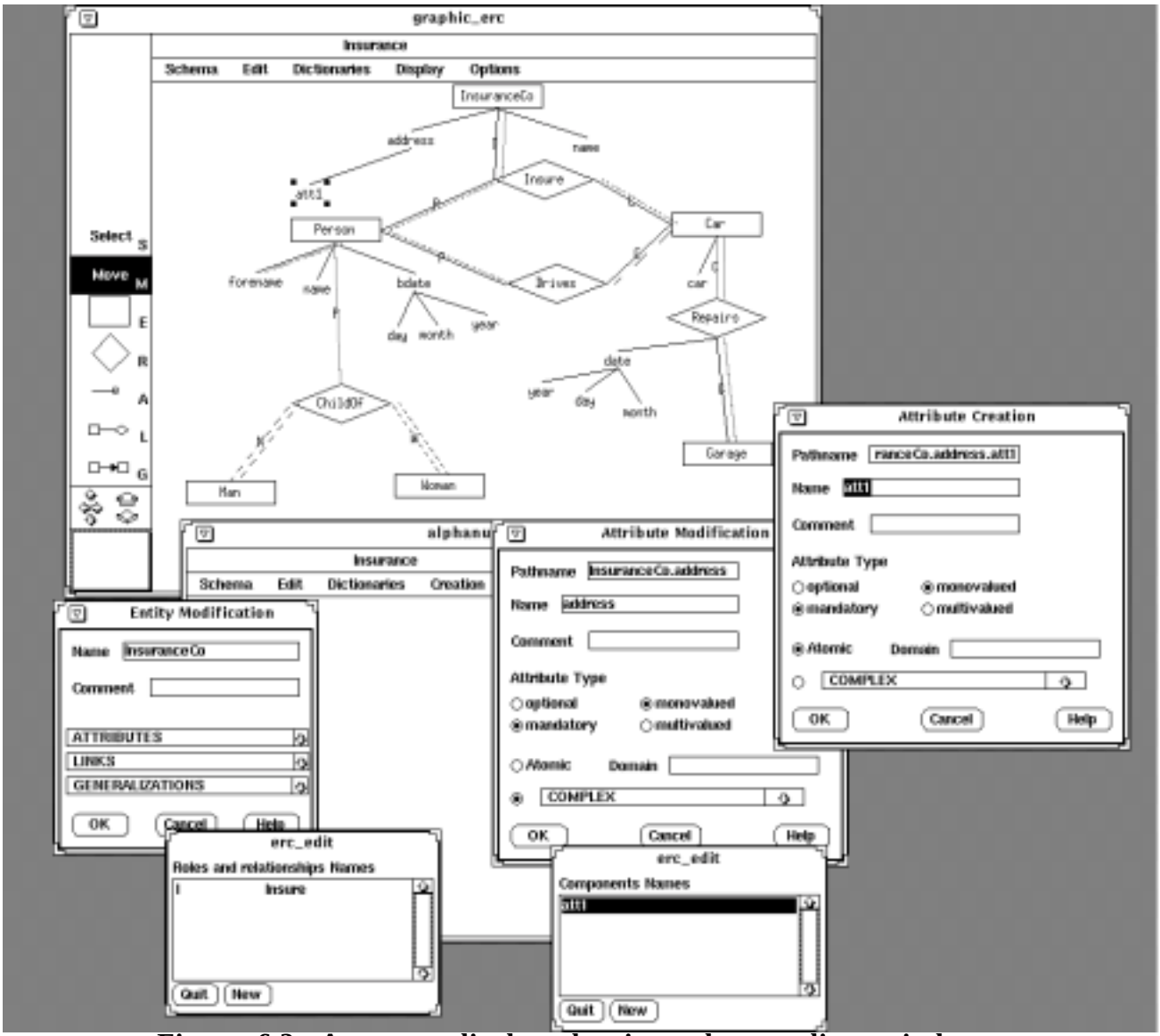


Figure 6.2: A screen display showing schema editor windows
(after creation of a new attribute, whose default name is att1)

This kind of flexibility is sometimes criticized as being confusing for users. We believe it might indeed be confusing if the alternatives appear within a single context, with users not having a criteria to choose in between. On the contrary, if the alternatives are provided along different paths, it avoids the burden of explicitly moving from the context they are in to the context which provides the desired function. In our case, users can create objects as they navigate through the schema, without having to go to the Creation menu.

Reusability allows users to reuse definitions of objects in the current schema or in another schema. Cut, copy and paste operations may be used to move an object (i.e. disconnect it and connect it elsewhere), delete it, or create a similar object elsewhere (if needed, because of uniqueness rule, the name of the object is automatically updated). The object here may be a single object (an entity type, an attribute, ...), or a collection of objects (for instance, a set of attributes may be copied from various existing objects and in one shot attached to an entity type), or a subschema (a set of entities, attributes, relationships and generalizations where the latter two must include the objects they link). A duplicate operation is also provided and creates an object identical to the original one (but with a different name) and bearing the same connections.

Finally, backtracking is supported through undo and redo operations. This allows users to recover from erroneous actions and restore the previous state. Typically, if a user clicks on a Cancel button instead of the nearby OK button, all actions performed on the object would be lost. By undoing the erroneous click, (s)he will get a second chance.

6.3 Schema browsing

The current version of SUPER supports schema browsing. Users may scroll the schema diagram to display the desired part of it. The alphanumeric mode allows schema browsing by navigation from one object to another through existing connections in between. This navigation may use object boxes (as shown in figure 6.2) to allow user to see all informations about the objects on the path.
A similar navigation may also be performed using a simultaneous display of the various dictionary lists. For instance, the selection of an entity type in the entity types list will automatically display its attributes, relationships and generalizations/specializations, if any, in the corresponding list. However, the only information users get in such a navigation are the names of the objects. To know more about a specific object, users have to click on its entry in the appropriate list, to activate its object box (the information contained in the object box is then only available for inspection, to prevent conflict between different actions on the same object).

7. Graphical manipulation languages

SUPER includes two data manipulation languages which both allow querying and updating an ERC+ database: a data browser and a query editor. Section 7.1 gives a short presentation of the data browser, while section 7.2 describes in more details the query editor.

7.1 The data browser

The SUPER browser provides two modes for data visualization. In the forms-based mode, occurrences are presented with forms-like representations of the corresponding object or relationship type, while in the graphical mode they are presented with entity-relationship-like diagrams showing the occurrences currently being examined. These occurrences can be directly manipulated by the user.

Regardless of whether forms or diagrams are used, each browsing session has an associated state that indicates if the session is in the query mode or in the update mode. In query mode, it is possible to browse the occurrences of a population (one at a time), to examine the values of their attributes and to move from an occurrence to another one belonging to a related population (related via a role or a multi-instanciation link). In the update mode, occurrences can be modified, deleted and created. Anytime during a session the user can switch between query and update mode with a click on a button . For instance (s)he can start browsing the database in the query mode, look for some data to be updated and switch to the update mode when the desired data is found.

7.2 The query editor

This section discusses the features concerning query formulation in the SUPER query editor. The steps which compose this process are the following:

- Selecting the query subschema: the portion relevant to the query is extracted from the database schema.
- Creating the query structure: the subschema is transformed into a hierarchical structure
- Specifying predicates: predicates are stated on database occurrences, so that only relevant data is selected.
- Formatting the output: the editor is provided with data items to be included into the structure of the result;
- Displaying resulting data.

The whole process may be rather complex, and therefore difficult to master for novice users. As these users are the main target of visual interfaces, we believe that visual query languages should take advantage of the above multistep structure. Indeed, clear separation between the steps alleviates users' mental load and improves the chances of correct formulation. The sequence of steps is logically meaningful: for instance, predicates cannot be defined before the query subschema is determined. Users can any time modify any stated part of the query to correct or refine the current formulation. SUPER implements step separation, by using specific windows for the different steps, as presented in the following.

7.2.1 Selecting the query subschema

This step corresponds to an "open subschema ..." command in textual languages. It configures the schema to contain only those objects which are involved in the query. The diagram corresponding to a schema is displayed in a read-only window, called the **database schema** (DBS) window.
The query subschema is extracted through a sequence of "point and click" specifications. To speed up this process, the semantics of the clicks can be tailored either as "keep" or as "delete" the designated object. Implicit designation is sometimes used: gql/ER [Zhang 83], for instance, automatically adds to the query subschema the path in between two selected objects (in case of multiple paths, the "most likely" one is chosen). QBD [Catarci 88] uses a similar technique, which can be refined with additional constraints (on the length of the path, for instance). It also allows predicates on attribute names to select all entity types with such attributes.

In SUPER, the "point and click" specifications copy objects in a second window, called the **working schema** (WS) window. The user may choose between two modes:

- the most usual one is a traditional Copy-Paste that transfers previously selected objects into the WS without relating them to objects already in the WS;
- in the second mode, called Expand, users select a start entity type in the WS and then click on objects in the DBS window for copying them in the WS and reconnecting them to the start entity type.

Some automatic selection is embedded in SUPER. If the user clicks on a role, the complete relationship type is transferred in the WS. If the relationship is binary, the start entity type changes to the new one. Clicking on a distant object is possible if there is a smallest path to the object. For instance, the user cannot click on a relationship which has two roles leading to the start entity type.

7.2.2 Restructuring the query subschema

Once a query subschema is defined, proper query formulation may start. However, some interfaces introduce an additional step, to transform the subschema into a specific pattern. In [Elmasri 85b] the query subschema is transformed into a hierarchical structure. The root of the hierarchy is selected by user. [Larson 86] follows the same approach, but the transformation is complemented with a generation of nested forms, visualizing the hierarchical structure. QBD provides a query-like transformation language for schema modification.
Graphical data manipulations in SUPER are based on the underlying ERC+ algebra, whose operators produce entities as results. These results are syntactical trees whose root is an entity with constraints expressed as predicates. In [Auddino 91] we discuss the use of tree representations of queries.

In our editor, the user identifies the root of a query hierarchy. A graph in a tree is transformed by first removing cycles. The removal of cycles could not be an automatic process. There are as many possible interpretations as there are duplications of entity types involved in the cycle. Such an ambiguity is not acceptable. As the editor cannot infer which interpretation is the intended one, the users should explicitly direct the transformation to be performed.
In the SUPER query editor, the user can break cycles by removing some vertices or some nodes of the graph or by disconnecting some links. Disconnection means that the designated link is detached from the linked entity type and attached to a (newly automatically created) copy of that entity type. Figure 7.1 shows the WS window before disconnection. Disconnecting the Drives---Car link produces the diagram in figure 7.2.

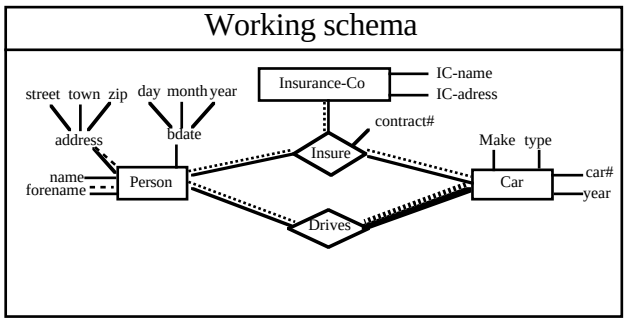


Figure 7.1. Before disconnection

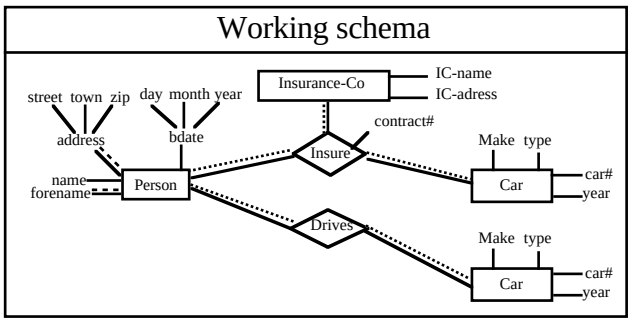


Figure 7.2. After disconnection

If the user wants to express in the query some information about cars, the editor cannot determine where to attach the Repairs relationship type (figure 7.3). It is up to the user to direct the editor to "unify" the two Car entity types (s)he designates. The diagram resulting from unification is shown in figure 7.4.

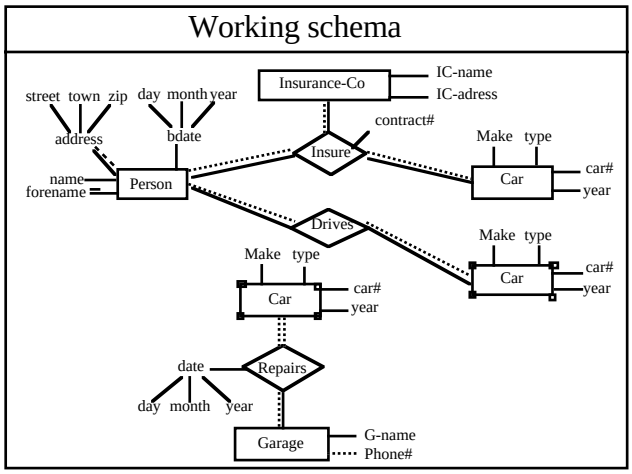


Figure 7.3. Before unification

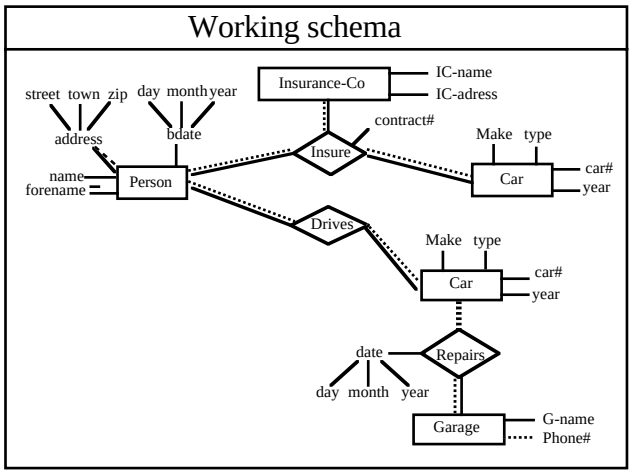


Figure 7.4. After unification

Disconnection and unification provide most of the functionality's needed for the proper definition of hierarchical queries over an ERC+ schema. Another facility, pruning, is used to remove objects (attributes, entity types, ...) which are not used in the query, i.e. appearing neither in the format of the result nor in a predicate. There are some additional facilities like the product or union to create an artificial link between two entities.

7.2.3 Specifying the predicates

Once the user has created a correct query structure in the WS, SUPER build the corresponding hierarchy as a single entity type, with all other informations as attributes. This resulting structure is displayed in a third window, called the **selection window** (SW - an example may be seen in figure 7.6). If the resulting structure is not what the user expected, (s)he can make the appropriate modifications in the WS.

If the resulting structure is correct (from user's viewpoint) (s)he will proceed with the specification of predicates. Predicates against complex objects may be rather clumsy. For the simplest ones (comparison of a monovalued attribute with a constant) a graphical counterpart may easily be defined.
A simple specification technique is to click on the attribute, select a comparison operator from a menu, and finally type the value or choose one from a list. For complex predicates (involving several quantifiers, for instance), there might be no simple way to express it graphically. Menus are sometimes used for syntactic editing of predicates ([Elmasri 85b] or ISIS [Goldman 85]). In gql/ER, QBE-like forms are used to specify conditions on the selected nodes. Only a few interfaces allow the use of a graphical formalism for expressing predicates (see, for instance, Pasta-3 or SNAP).

In this paper we do not want to discuss about the best graphical solution for predicate specification. Rather we focus on functionality's. Predicates are expressed on the hierarchical structure (entity type) resulting from the previous step. A predicate is any logical expression involving attributes of the final entity type. The predicate implements the selection operator if it is attached to the root or the reduction operator if it is attached to an attribute. The second step of the definition of a predicate is the definition of its domain. The domain is the set of quantified variables. In the following, we show some examples of predicate specification.

Consider figure 7.6 hereinafter. Assume the condition "= 1944" is associated to the year attribute. Year is a monovalued mandatory attribute. The interpretation of the predicate is straightforward: the query will select as resulting entities only those where year=1944.
Assume the condition "= Paris" is associated to the town attribute. Town is a 1:1 component of the 1:n attribute address. Here a quantifier is needed to specify the condition the selected entities have to satisfy. The existential quantifier will select entities for which there is one address value such that town=Paris, while the universal quantifier will select those for which all address values satisfy the condition town=Paris. As in Pasta-3, SUPER assumes (by default) the existential quantifier if none is explicitly defined by the user.
Consider now the selection window shown in figure 7.15 hereinafter. Assume the condition "= Morris" is associated to the G-name component of the Insure-2 attribute. G-name is two levels of multivaluation below the entity type (two multivalued attributes, Insure-2 and repairs, lay along the path leading to G-name). Consequently, two quantifiers are needed to define the query: one to specify whether every repair, or just one, has to be made by Morris, the second to specify whether this condition has to hold for every, or just one, car in Insure-2.
Finally, considering again figure 7.6, the user might want to select entities such that they have one address with town=Paris and one address with town≠Paris. To state this predicate, the user needs to designate two different values for address. In a textual language, this is performed by using two variables associated to the same attribute. In SUPER, the user will ask for duplication of the attribute, which will allow the specification of both predicates and of the logical connectors in between (and, or, not). Generally speaking, every attribute which appears several times in one or several predicates must be represented as many times in the selection window.

Evaluation of the predicates can be done in any order. Each intermediate step usually builds a potential query that can be interpreted (a syntactic validation) and executed (a semantic validation on an existing database) in the fourth window.

7.2.4 Formatting the output

By default, the selection window defines the structure of the resulting entity type. However, the users may wish to discard some of the attributes, which have been kept up to now only because of some predicate to be defined on them. The selection window has to provide for a "hide" (or, conversely, "show") operation, to define which attributes are to be discarded (or kept in). The hiding (or the showing) of a complex attribute also hides (or shows) all its component attributes.

7.2.5 Displaying resulting data

The last phase is the display of instances representing the result of the query. SUPER displays resulting entities according to their hierarchical structure, into a nested tabular form or into an attribute tree form. To that extent, a fourth window, the **result window** is used (Figure 7.11). Users can choose between the two kinds of presentation through a switch mode radio button.

Relational interfaces display occurrences in a tabular form. SNAP provides the user with the choice between the tabular format and a NF2 format, where occurrences are arranged into "buckets". [Larson 86] uses the nested forms representation to display the results of data browsing. GUIDE [Wong 82] and VGQF [McDonald 84] allow the user to choose among different formats. In OdeView [Agrawal 90] complex objects can be displayed through a text or picture representation; the user can click on buttons to display all related objects.

7.2.6 Additional facilities

Additional functionality's supported by SUPER allow to store queries for later reuse or modification, as well as evaluation of partial queries. The latter is useful for query debugging. Suppose the user is confronted with a result different from what was expected, without recognizing what is the problem within the formulation. The query can be broken into two or more separate queries by disconnecting some links. Independent evaluation of the sub queries can be performed to identify what changes have to be made. After this refinement, the original, corrected query can easily be rebuilt by unification of the duplicated entity types created by the previous disconnection.

7.2.7 A sample query

This section illustrates the process of query formulation in SUPER. Let us assume the user wants to formulate the following query on the schema of figure 6.1.

Select name and address of people who insure a 1984 Ford

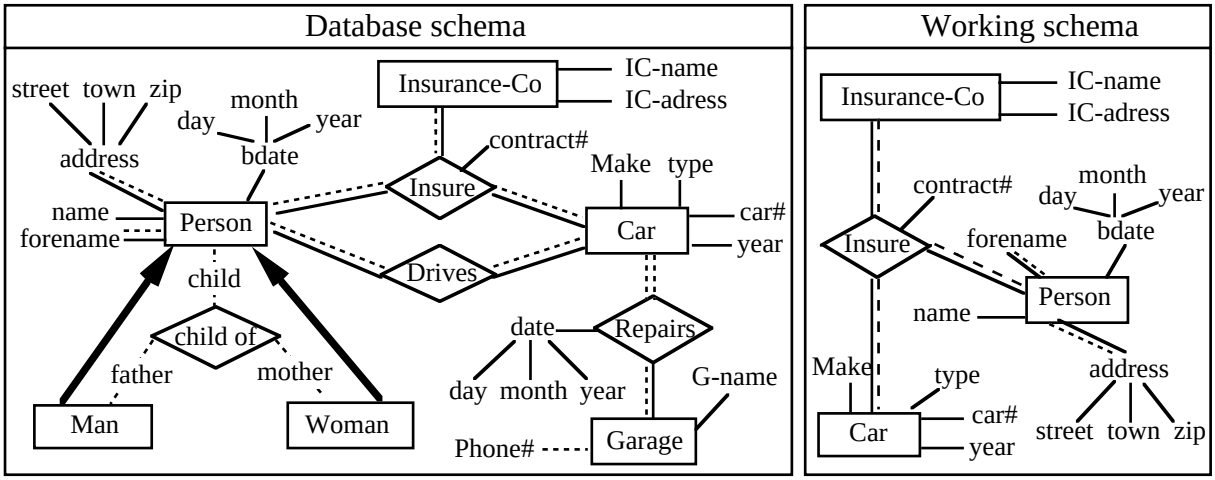


Figure 7.5. Query editor windows showing query subschema definition

The corresponding diagram will be displayed in the database window. The user begins by picking the relationship type Insure, that will be copied into the WS window, together with the linked entity types (Person and Car) and all their attributes (Figure 7.5).
Next, assume the user designates Person as the root of the hierarchy. Figure 7.6 shows the contents of the selection window.

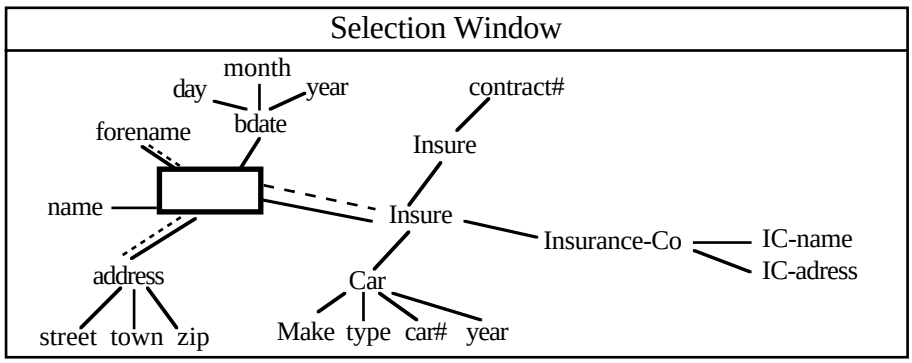


Figure 7.6. Resulting hierarchical structure, showing the root as an empty box

This structure includes many attributes the user is not interested in. Consequently, (s)he will return to the WS window and prune unnecessary objects. The attributes name and address of Person are needed for result display, while Make and year of entity Car will be used for predicate specification. Pruning will change the contents of windows, as shown in figure 7.7.

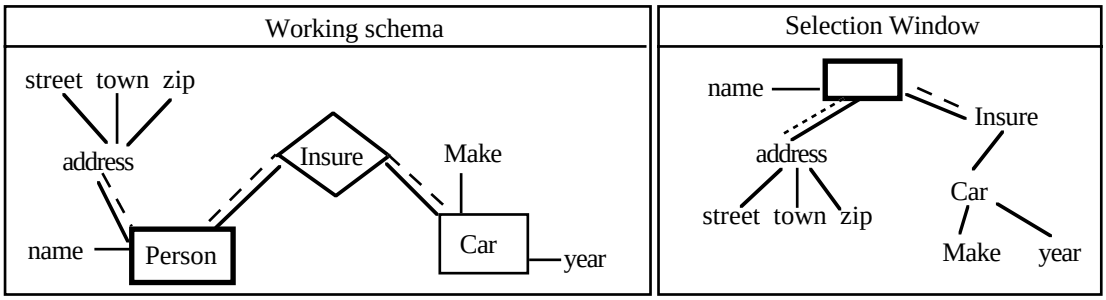


Figure 7.7. The updated query subschema and corresponding hierarchical structure

The definition of a predicate is made through a predicate box displayed in the selection window (Figure 7.8).

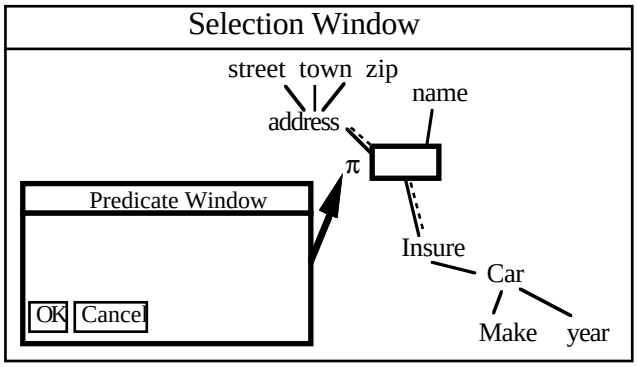


Figure 7.8. The initial state for predicate definition

The user designates the attributes (Make and year) involved in the predicate: Make and year. Figure 7.9 shows the selection window after designation of Make. As the Insure attribute (to which Make belongs) is multivalued, a modifiable "exists Insure" clause is automatically generated.
While designating the second attribute, year, the user also has to specify the logical connector between the two predicates. The predicate box now contains all the necessary quantified attributes. The specification is completed by entering the appropriate values: Ford for Make and 1984 for year (Figure 7.10).

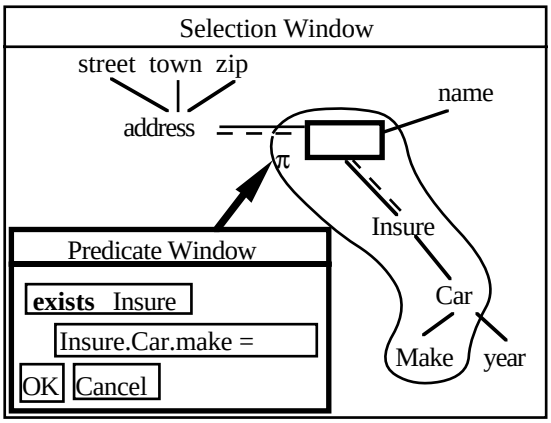


Figure 7.9. SW after designation of Make

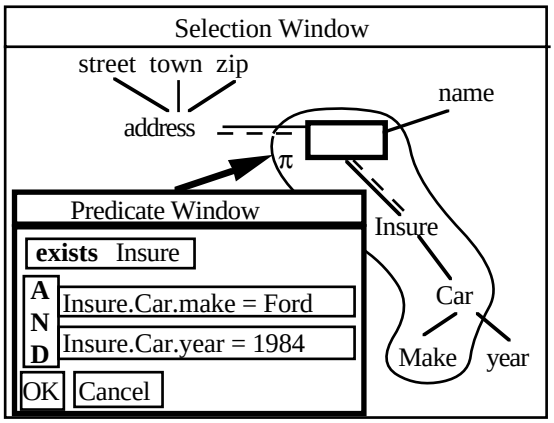


Figure 7.10. Final state of the SW

Then, the user hides the attribute Insure which does not belong to the result, and the query is ready for evaluation. Figure 7.11 shows the resulting occurrences.

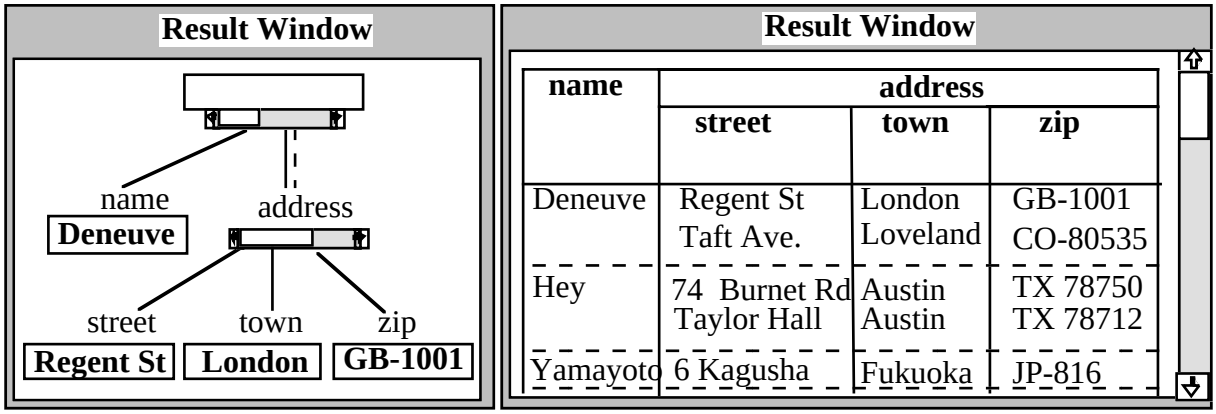


Figure 7.11. Result of the evaluated query in the attribute tree form and in the nested tabular form

Suppose now that the user wants to proceed with the specification of the following query, very similar to the previous one:

Name and address of persons who insure a 1984 Honda, and who also insure a car that has been repaired in the "Morris" garage.

The user will import in the WS window the additional information in the DBS window needed (the relationship type Repairs). The result is shown in figure 7.12.

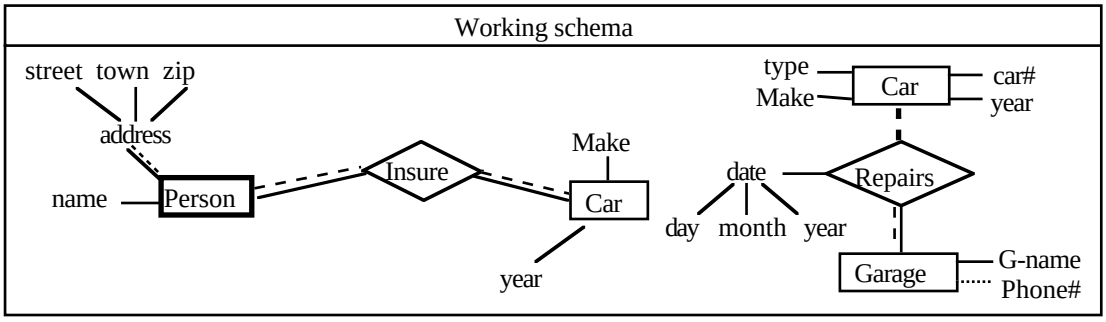


Figure 7.12: starting a query modification

Next, the user will unify the two Cars entity types and prune unneeded objects (figure 7.13). Then, (s)he can redefine the first predicate to select 1984 insured Honda (figure 7.14).

The second condition (insuring a car that has been repaired in the "Morris" garage) involves the Insure attribute, independently from the first condition, as Insure is multivalued. Its specification calls for the duplication of Insure. The predicate can now be expanded to include the new condition. By default, Insure-2 and its repairs component have been existentially quantified. The final situation is shown in figure 7.15.

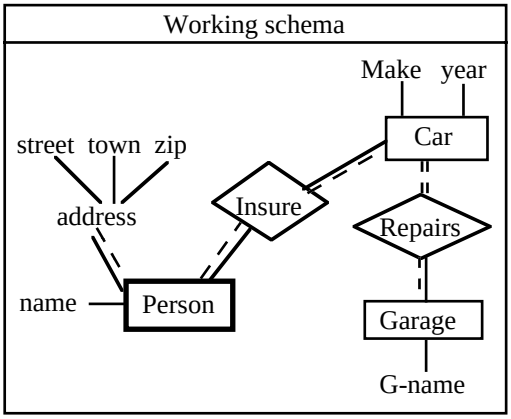


Figure 7.13: Keep useful objects

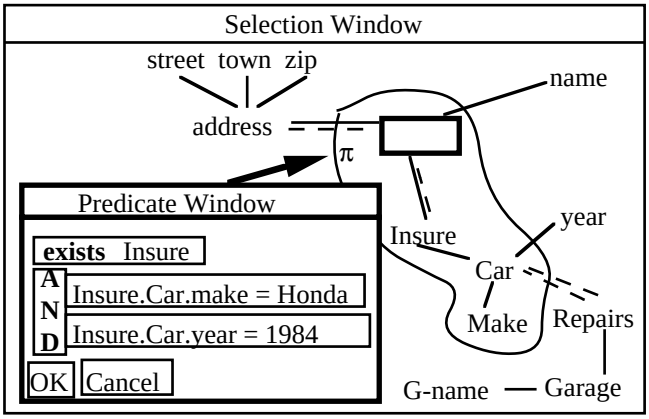


Figure 7.14: Define a predicate

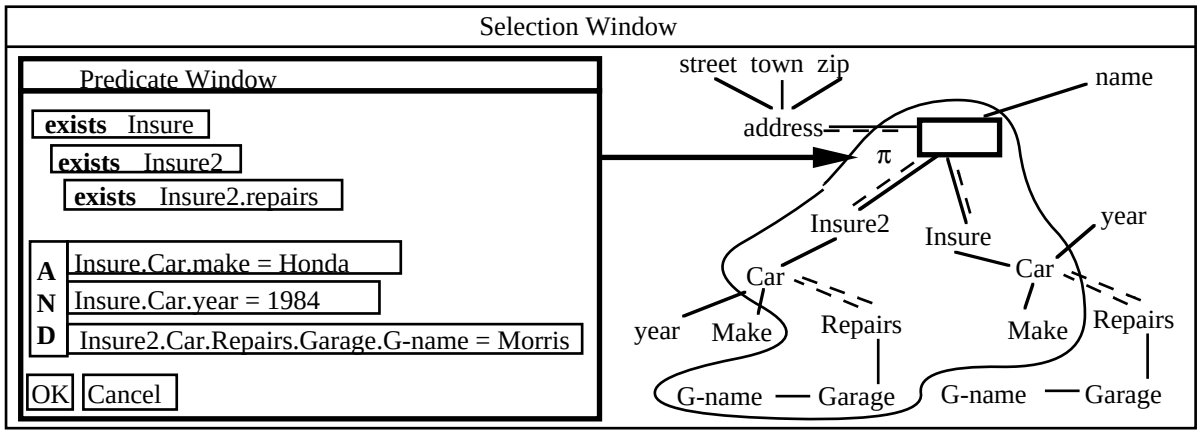


Figure 7.15: The new query ready for evaluation

8. Describing application dynamics

The previous sections discussed application requirements in terms of structural modeling and classical data manipulation operations. This section considers requirements for the description of application dynamics and briefly sketches the steps necessary to integrate a process model of system dynamics with the ERC+ model.

Traditionally, the focus in this area has been on capturing the events which are significant in the application's life cycle. Events trigger operations on the database, according to pre-conditions; operations produce post-conditions which eventually generate new events. Emphasis in this approach is on the global control structure of the information/activation flow between the processes composing the application. The Petri net formalism is a powerful model to describe the control structure of concurrent systems, and it has been largely used as a formal vehicle to express this view of the dynamics [Peterson 82, Reisig 85]. Unfortunately, for the basic place/transition model the representation of complex systems often leads to very large nets which are difficult to analyze. To avoid this problem, high-level Petri nets have been developed, and will be briefly illustrated here by making reference to one of them, the Predicate-Transition (PrT) net model described in [Genrich 81].

The main difference between the PrT net model and the basic Petri net model is that tokens in PrT nets can be structured objects and the transition firing is controlled by imposing conditions on the token values.
A PrT net consists of the following constituents [Genrich 81]:

1) a directed net (**P**,**T**;**F**), where **P** is the set of predicates ("first-order" places), **T** is the set of transitions, **F** is the set of arcs, and:

$\mathbf{P} \cap \mathbf{T} = \emptyset$, $\mathbf{P} \cup \mathbf{T} \neq \emptyset$, $\mathbf{P} \otimes \mathbf{T} \cup \mathbf{T} \otimes \mathbf{P} \supseteq \mathbf{F}$

$\bullet T = \{P | (P,T) \in \mathbf{F}\}$ and

$T \bullet = \{P | (T,P) \in \mathbf{F}\}$

are called the preset and postset of $T \in \mathbf{T}$, respectively.

2) a structure Σ, consisting of some sorts of individuals together with some operations and relations;

3) a labeling of all arcs with a formal sum of tuples of variables, whose length n is the 'arity' of the predicate connected to the arc. The zero-tuple indicating a no-argument predicate (an ordinary Petri net place) is denoted by the special symbol ¢;

4) an inscription on some transitions, being a logical formula built from the operations and relations of the structure Σ. Variables occurring free in a formula have to occur at an adjacent arc;

5) a marking M_0 of predicates of **P** with formal sums of n-tuples (items) of individual symbols;

6) a function K, which assigns to each predicate $P \in \mathbf{P}$ an upper bound for the number of copies of the same item which it may carry. K(P) is called the capacity of P;

7) the transition rule: each element of **T** represents a class of possible changes of the markings of the adjacent predicates. Such a change consists of removing/adding copies of items from/to the adjacent places according to the expressions labeling the arcs. It may occur whenever, for an assignment of individuals to the variables which satisfies the formula attached as inscription to the transition, all input predicates carry enough copies of proper items, and for no output predicate the capacity K is exceeded by adding the respective copies of items. The occurrence (firing) of T changes the marking M into a new marking M'; this fact is denoted by: M [T > M'].

The structure Σ can be specified by means of a first-order language L which describes the individuals of Σ, their properties and relations. Let Ω be a set of n-ary operators (function symbols), Π a set of n-ary predicates (relation symbols), and V a set of symbols (disjoint from Ω and Π) which will be used as variables. The language L consists of terms and formulas which are built in the following way:

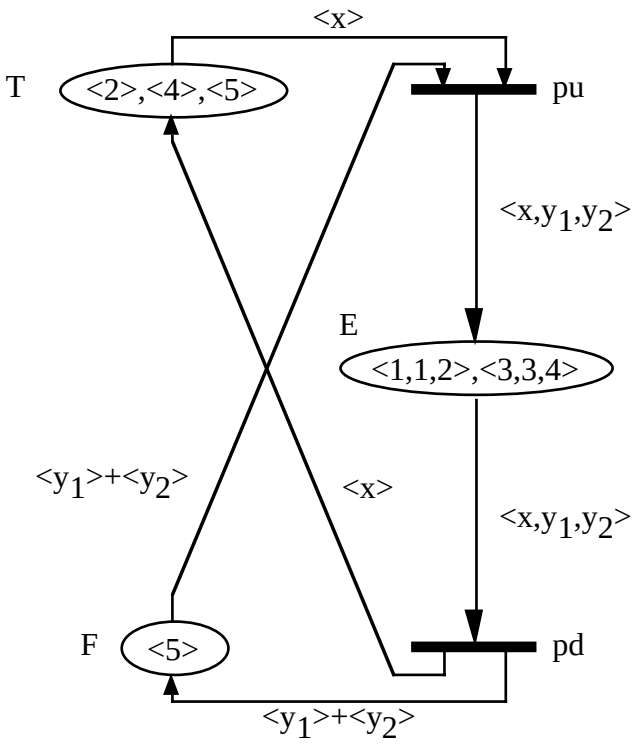
Terms.
a) a variable is a term,
b) if **f** is an n-ary operator and $v_1,...,v_n$ are terms, then $f(v_1,...,v_n)$ is a term,
c) no other expression is a term (remark: constants, i.e. 0-ary operators, are terms).

Formulas. A (well-formed) formula is defined as:
a) if v_1 and v_2 are terms, then $v_1 = v_2$ is an atomic formula (or, simply, an atom),
b) if P is an n-ary predicate and $v_1,...,v_n$ are terms, then $P(v_1,...,v_n)$ is an atom,
c) if p_1 and p_2 are formulas, then not p_1 and (p_1 or p_2) are formulas,
d) if x is a variable and p a formula, then $(\forall x)p$ is a formula,
e) no other expression is a formula (remark: the connectors and, $\rightarrow$, $\leftrightarrow$ and $\exists$ are derived from not, or, and $\forall$ in the usual way).

To illustrate this approach, let us consider a simple example, the well known Five Philosophers (FP) problem [Peterson 82]:

FP: Five philosophers are seated at a round table, eating spaghetti. Between each philosopher is one fork. To eat, a philosopher must pick up both the fork on his left and the fork on his right. When a philosopher is not eating, he is thinking (and vice versa). The problem is to synchronize the activities of philosophers since they cannot eat all at the same time because of lack of forks.

Figure 8.1 shows the representation of the FP problem in the PrT net model.



Basic domain: D = {1,2,3,4,5}

Operators:

l(i) = i	left-fork function
r(i) = (i+1) mod 5	right-fork function

Predicates:

$T \subseteq D$	thinking philosophers
$F \subseteq D$	free forks
$E \subseteq T \otimes F \otimes F$	eating philosophers with related forks

Transistions:

pick-up	pu [y_1 = l(x) and y_2 = r(x)]
put-down	pd

Initial marking:

μ_o = {T:{<2>,<4>,<5>}, E:{<1,1,2>,<3,3,4>, F:{<5>}

Figure 8.1: The PrT description of the FP problem.

Starting from the state illustrated in figure 8.1, the transition pd can occur with the substitution:

{x/3,y_1/3,y_2/4}

which generates the new marking:

$$\Delta_1 = \{T:(<2>,<3>,<4>,<5>),\ E:(<1,1,2>),\ F:(<3,4,5>)\}$$

As shown by the PrT net of figure 8.1, while the dynamic behavior of the system is expressed in a simple way by using a PrT net, the effective specification of the system requires that the structure Σ and the transaction inscriptions must be explicitly defined. In general, this is not a simple task!

Moreover, real-life systems are usually managed by means of large database systems which store data and complex relationships between them. How can a PrT system specification be related to the data structures stored into the database which support the system operation?

Our aim is to propose a model to provide an effective solution for this problem. The model relates in a simple way static and dynamics specifications of the system under development. It integrates a process model (which is basically the PrT model) to represent the system dynamics and the ERC+ data model to describe data stored into the database. For the sake of clarity we will henceforth use the term ERC++ to denote the model resulting from this integration.

To illustrate the above discussion, consider the classical Five Philosophers problem. It can be described by means of the ERC+ data schema shown in figure 8.2, where the initial database state (db-state) DBo corresponds to the initial marking Δ_o of figure 8.1. The relationships Left and Right give, respectively, the left and the right fork of a philosopher.

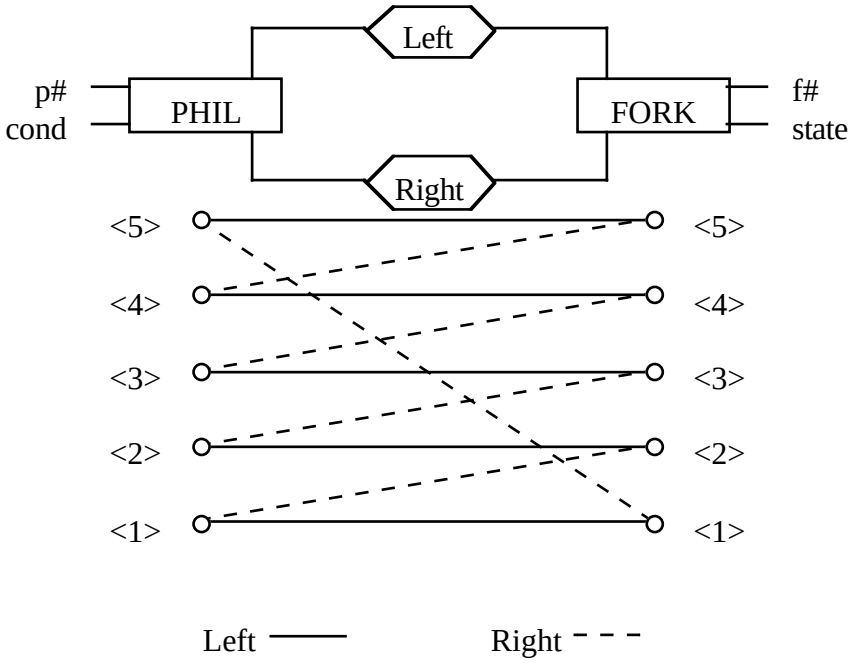


Figure 8.2: The ERC+ specification of the FP database.

In the ERC++ model, the system behavior is specified with PrT nets in which predicates and transitions are described by means of formulas in the ERC+ algebra (or the ERC+ calculus). Such kind of PrT nets will be called ERC++ nets. The ERC++ net which expresses the dynamic behavior of the FP system on the database specified in figure 8.2 is described by means of the ERC+ algebra formulas in figure 8.3 or the equivalent ERC+ calculus formulas of figure 8.4 (only formulas which specify predicates and transitions in the PrT net of figure 8.1 are reported).

T: **select** [cond='thinking'] PHIL
E: **select** [cond='eating'] PHIL
F: **select** [state='available'] FORK

pu: (T **sel-r-join** (Left, F)) **sel-r-join** (Right, F)

Figure 8.3: The ERC++ algebra specification of the FP system.

T: $\{ x \mid x \in \text{PHIL} \wedge x.\text{cond}=\text{'thinking'}\}$

E: $\{ x \mid x \in \text{PHIL} \wedge x.\text{cond}=\text{'eating'}\}$

F: $\{ y \mid y \in \text{FORK} \wedge y.\text{state}=\text{'available'}\}$

$\text{pu} : (x \in T \wedge y_1 \in F \wedge y_2 \in F \wedge \text{Left}(x, y_1) \wedge \text{Right}(x, y_2))$

Figure 8.4: The ERC++ calculus specification of the FP system.

The OO approach moves the focus from the application as a whole to its objects, seen as independent interacting agents. Dynamics is expressed as a sequence of messages passing from one object to another one (hence the expression "active objects"). This definitely facilitates the control over the integrity of each object in the database. The designer of an object type has to be fully aware of all possible stimuli the objects may have to respond to. (S)he can then define the appropriate methods to be attached to the object type. As no agent external to the associated methods can modify the objects, object integrity is guaranteed just by designing each method to be integrity-preserving.

The counterpart of gaining more accuracy and control over the objects is that it becomes difficult to get a global comprehension and overall control of the application, and to enforce integrity rules spanning over several objects (global consistency). It is hard to figure out, by looking at methods in

objects' definitions, what the application does and how it works. It is also hard to figure out, when designing a new application, which objects and methods may be reused. It is often the case that a designer defines new methods for a new application, just because (s)he does not know that a similar method already exists or because of slightly different requirements. Also, an attempt to reuse existing methods often results in the creation of new object types and inheritance structures which are pure programming artifacts, with no counterpart in the real world.

To sum up, methods bear clear advantages, but are uneasy to use and do not suffice by their own to cope with all aspects of dynamics. Therefore, the traditional and the OO approaches should be considered as complementary, rather than an alternative to each other. The ERC++ model is planned to support both.

For global dynamics (i.e., description of the application life-cycle), it will provide support based on high-level Petri nets, as sketched above. To this end, a user-oriented ERC++ language based on the ERC+ algebra is under development. The ERC++ language will provide facilities to specify (in a more intuitive way than algebra or calculus) predicates and transitions of the PrT control net which describe the system evolution.

On the other hand, the ERC++ language will equally provide facilities for attaching specific methods to application objects and for using such methods to describe the database state change related to a given transition. General methods for ordinary access to the information in the objects will also be supported.

Designing methods is one of the most difficult tasks in building OO applications. There is little methodological support, or tools, available. In principle, methods have to be developed to express the intrinsic behavior of a set of objects, where with intrinsic behavior we mean the set of operations and request protocol representing a program component common to different applications of the information system. It must be noted that intrinsic behavior cannot be discovered looking at one application at a time; in fact, the extraction of common components is a methodological problem which must be taken into account in an object-oriented methodology for information system design. To alleviate this task, SUPER is investigating the concepts needed to build a tool for automatic generation of methods specifications from high-level descriptions of different application requirements in terms of data processing.

9. Conclusion

A lot of research results have documented the advantages of object oriented concepts for tackling the design of large and complex software systems. Experience gained from using OO models to design database applications has clearly stated the need to combine the key features of object orientation with concepts for representing explicit relationships among objects and for expressing the integrity constraints which are necessary to capture the semantics of database systems. We have proposed in this paper an object+relationship approach, called ERC+, which has been designed to fully support database application requirements. To achieve this aim, ERC+ starts from a well-accepted semantic modeling paradigm (the ER approach) and improves its modeling capabilities with constructs for representing complex objects, and OO features (the methods) and Petri net based process descriptions for representing behavioral aspects. Moreover, ERC+ provides generic data manipulation languages for user-DBMS interaction: theoretical languages (an algebra, a calculus) support definition of user-oriented languages (SQL-like, graphical).

Beyond the definition of a modeling approach, there is also a need for an integrated set of tools to support the methodology. To that purpose, we have defined and are currently implementing a design environment called SUPER to support the ERC+ modeling approach. The characteristic features of SUPER are:
- a graphical schema editor for the definition of complex objects. The schema editor provides a palette of graphical symbols corresponding to each of the concepts of the ERC+ model: entity types, relationship types, etc. It allows a designer to build a schema through click and pick selections.
- a graphical query editor for data manipulation. The query editor is a tool which allows graphical formulation of data manipulation operations that are automatically translated into expressions in the underlying ERC+ algebra. Graphical query formulation is achieved by: selecting a query schema, creating a query structure (a template), specifying predicates, specifying the structure of the query

results, and by displaying the resulting data. The user is allowed to perform these steps in any meaningfully consistent sequence.
- a data browser which allows users to navigate through the database, displaying the visited occurrences either within dynamic forms or as graphical diagrams similar to schema diagrams.

These tools have been implemented in C++. The underlying algebraic language is also implemented. Our current research focuses on the specification of a tool to incrementally design objects through view integration. This tool implements a methodology in which complex systems are built by first decomposing the application into a number of small manageable tasks. Next, a view corresponding to a partial perception of the global data description of the application is associated with each task. And finally the resulting views of the different tasks are integrated to build the global schema of the application.

References

[Abiteboul 89] S. Abiteboul 89], P.C. Fischer, H. J. Scheck Eds.: *Nested Relations and Complex Objects in Databases,* Lectures Notes in Computer Science 361, Springer Verlag, 1989

[Agrawal 90] R. Agrawal, N. H. Gehani, J. Srinivasan: *OdeView: The Graphical Interface to Ode*, ACM SIGMOD International Conference on Management of Data, Atlantic City, May 23-25, 1990, pp. 34-43

[Albano 91] A. Albano, G. Ghelli, R. Orsini: *A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language*, 17th International Conference on Very Large Data Bases, Barcelona, September 3-6, 1991, pp. 565-575

[Auddino 91] A. Auddino & al.: *SUPER: A Comprehensive Approach to DBMS Visual User Interfaces*, IFIP WG 2.6 2nd Working Conference on Visual Database Systems, Budapest, September 30-October 3, 1991, pp. 359-374

[Bancilhon 88] F. Bancilhon & al.: *The Design and Implementation of O2, an Object-Oriented Database System*, in Advances in Object-Oriented Database Systems, K.R. Dittrich Ed., Lecture Notes in Computer Science no 334, Springer-Verlag, 1988, pp. 1-22

[Bertino 92] E. Bertino, M. Negri, G. Pelagatti, L. Sbattella: *Object Oriented Query Languages: The Notions and the Issues*, IEEE Transactions on Data and Knowledge Engineering, Vol.4, No.3, June 1992, pp. 223-237

[Booch 91] G. Booch: *Object Oriented Design with Applications*, Benjamin Cummings, 1991

[Brodie 84] M. Brodie, J. Mylopoulos, J. Schmidt (Eds.): *On Conceptual Modelling*, Springer-Verlag, 1984

[Carey 88] M. Carey, D. DeWitt, S. Vandenberg: *A Data Model and Query Language for EXODUS* , ACM-SIGMOD International Conference on Management of Data, Chicago, June 1-3, 1988, pp. 413-423

[Catarci 88] T. Catarci, G. Santucci: *QBD: A Graphic Query System* , 7th International Conference on the Entity-Relationship Approach, Roma, November 16-18, 1988, pp. 157-174

[Chen 76] P.P. Chen: *The Entity-Relationship Model - Towards a Unified View of Data*, ACM Transactions On Database Systems, Vol.1, No 1, 1976

[Czedjo 90] B. Czejdo, R. Elmasri, M. Rusinkiewicz, D. Embley: *A Graphical Data Manipulation Language for an Extended Entity-Relationship Model*, IEEE Computer, Vol.23, No 3, March 1990, pp. 26-36

[Elmasri 85a] R. Elmasri, J. Weeldreyer, A. Hevner: *The category concept : an extension to the entity-relationship model*, Data & Knowledge Engineering, Vol. 1, n° 1, June 1985, pp. 75-116

[Elmasri 85b] R. A. Elmasri, J. A. Larson: *A Graphical Query Facility for ER Databases*, in Entity-Relationship Approach - The Use of ER Concept in Knowledge Representation, P. P. Chen ed., North-Holland, 1985, pp. 236-245

[Genrich 81] Genrich H.J., Lautenbach K.: *System Modelling with High Level Petri Nets*, Theoretical Computer Science 13 (1981)

[Goldman 85] K. J. Goldman, S. A. Goldman, P. C. Kanellakis, S. B. Zdonik: *ISIS, Interface for a Semantic Information System*, ACM SIGMOD International Conference on Management of Data, Austin, 1985, pp. 328-342,

[Hohenstein 89] U. Hohenstein *Automatic Transformation of an Entity-Relationship Query Language Into SQL*, 8th International Conference on the Entity-Relationship Approach, Toronto, October 18-20, 1989, pp. 309-327

[Jaeschke 82] G. Jaeschke: *Remarks on the Algebra of the Non First Normal Form Relations*, ACM SIGMOD International Conference on Management of Data, Los Angeles, March 1982

[Khoshafian 90] S. Khoshafian, R. Abnous: *Object Orientation - Concepts, Languages, Databases, User Interfaces*, John Wiley & Sons, 1990

[Larson 86] J. Larson: *A Visual Approach to Browsing in a Database Environment*, IEEE Computer, vol. 19, no. 6, June 1986, pp. 62-71

[Loizou 91] G. Loizou, P. Pouyioutas: *A Query Algebra for an Extended Object-Oriented Database Model*, International Symposium of Database System for Advanced Applications, Tokyo, April 1991

[McDonald 84] N. H. McDonald: *A MultiMedia Approach to the User Interface*, in Human Factors and Interactive Computer Systems, Y. Vassiliou ed., Ablex Publishing Corp., 1984, pp. 105-116

[Navathe 86] S.B. Navathe, R. Elmasri, J.A. Larson: *Integrating User Views in Database Design*, IEEE Computer, Vol.19, n° 1, January 1986, 50-62

[Parent 90] C. Parent, H. Rolin, K. Yétongnon, S. Spaccapietra: *An ER Calculus for the Entity-Relationship Complex Model*, in Entity-Relationship Approach to Database Design and Querying, F. Lochovsky ed., Elsevier Science Pub., 1990, pp.361-384

[Parent 92] C. Parent, S. Spaccapietra: *ERC+: an object based entity-relationship approach*, in Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development, P.Loucopoulos, R.Zicari Eds., John Wiley, 1992

[Pernici 90] B. Pernici: *Objects with roles*, IEEE/ACM International Conference on Office Information Systems, Cambridge, Mass., April 25-27, 1990, pp. 205-215

[Peterson 82] Peterson J.L.: *Petri Net Theory and the Modelling of Systems*, Prentice Hall, 1982

[Reisig 85] Reisig R.: *Petri Nets*, Springer-Verlag,1985

[Richardson 91] J. Richardson, P. Schwarz: *Aspects: Extending Objects to Support Multiple, Independent Roles*, ACM SIGMOD International Conference on Mangement of Data, Denver, May 29-31, 1991, pp. 298-307

[Rumbaugh 87] J. Rumbaugh: *Relations as Semantic Constructs in an Object-Oriented Language*, OOPSLA Conference, Orlando, October 4-8, 1987, pp.466-481

[Roth 84] M. Roth, H. Korth, A. Silberschatz: *Theory of Non First Normal Form Relational Databases*, TR-84-36 Department of Computer Science, University of Texas at Austin, December 1984

[Scholl 90] M.H. Scholl, H.-J. Schek: *A Relational Object Model*, 3rd International Conference on Database Theory, Paris, December 1990

[Spaccapietra 92] S. Spaccapietra, C. Parent: *Model Independent Assertions for Integration of Heterogeneous Schemas*, The VLDB Journal, Vol.1, No 1, July 1992, pp.81-126

[Sunye 92] M. Sunye: *CERQLE: un SQL entité-relation*, 7èmes Journées Bases de Données Avancées, Trégastel, September 15-18, 1992

[Vrbsky 89] S. V. Vrbsky, J. Liu and K Smith: *An Object-Oriented Approach to producing monotonically improving approximate answers*, Technical Report No. NVY N00014 89-J-1181 Dept. of Comp Science, Univ. of Illinois, Urbana-Champaign

[Wong 82] H. K. T. Wong, I. Kuo: *GUIDE: Graphic User Interface for Database Exploration*, 8th International Conference on Very Large Databases, Mexico City, 1982, pp. 22-32

[Zaniolo 83] C. Zaniolo, D. Maier: *The Database Language GEM*, ACM SIGMOD International Conference on Mangement of Data, San Jose, May 1983

[Zhang 83] Z. Q. Zhang, A. O. Mendelzon: *A Graphical Query Language fort Entity-Relationship Databases*, in Entity-Relationship Approach to Software Engineering, Davis et al. eds., North-Holland, 1983, pp. 441-448